NWC TP 7004

Tutorial on Using LISP Object-Oriented Programming for Blackboards: Solving the Radar Tracking Problem

by
P. R. Kersten
Research Department
and
Professor A. C. Kak
Robot Vision Laboratory
School of Electrical Engineering
Purdue University

AUGUST 1989

NAVAL WEAPONS CENTER
China Lake, CA 93555-6001

Naval Weapons Center

FOREWORD

This report describes an artificial intelligence architecture used to solve the radar tracking problem. The research described was performed at Purdue University during long-term training between August 1986 and August 1988. Continuing support into fiscal year 1989 was provided by 6.1 funds from individual research and development funds from the Office of Naval Research.

Approved by
R. L. DERR, *Head*
Research Department
13 June 1989

Under authority of
J. A. BURT
Capt., U. S. Navy
Commander

Released for publication by
G. R. SCHIEFER
Technical Director

NWC Technical Publication 7004

Published by ..... Technical Information Department
Collation ..... Cover, 46 leaves
First printing ..... 50 copies

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Public release; distribution is unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) NWC TP 7004	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Weapons Center	6b OFFICE SYMBOL (If Applicable) / 7a NAME OF MONITORING ORGANIZATION
6c ADDRESS (*City, State, and ZIP Code*) China Lake, CA 93555-6001	7b ADDRESS (*City, State, and ZIP Code*)
8a NAME OF FUNDING/SPONSORING ORGANIZATION Office of Naval Research	8b OFFICE SYMBOL (If Applicable) / 9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (*City, State, and ZIP Code*) Arlington, VA 22217	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO See back of page / PROJECT NO / TASK NO / WORK UNIT NO

11 TITLE (*Include Security Classification*)

TUTORIAL ON USING LISP OBJECT-ORIENTED PROGRAMMING FOR BLACKBOARDS: SOLVING THE RADAR TRACKING PROBLEM (U)

12 PERSONAL AUTHOR(S)

Kersten, P. R., and Kak, A. C.

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Interim	From 86 Aug To 88 Aug	1989, August	96

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (*Continue on reverse side if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	Artificial intelligence; Blackboard; Radar tracking problem; Object-oriented programming; Goal-driven blackboard

19 ABSTRACT (*Continue on reverse side if necessary and identify by block number*)

(U) The blackboard (BB) problem-solving model is an important problem-solving paradigm that has been applied to diverse environments from robot planning to signal processing. This model permits parallel processing with cooperation of problem-solving activities occurring via a centralized database. Goal-directed BBs form a key building block for distributed problem-solving systems where the problem is partitioned in space or time. The BB solves subproblems and is an important part of the distributed control.

(U) This report serves as a tutorial on how to use object-oriented programming in LISP to program a BB by explaining the important aspects of our radar tracking BB system. (Contd. on back)

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION	
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☒ DTIC USERS	Unclassified	
22a NAME OF RESPONSIBLE INDIVIDUAL P. R. Kersten	22b TELEPHONE (*Include Area Code*) (619) 939-3124	22c OFFICE SYMBOL Code 3807

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

19. The report demonstrates how flavors (object-oriented programming in Franz is carried out via flavors) can be used for this programming. Different approaches to object-oriented programming share considerable similarity, so this report should help even those readers who may not wish to use flavors.

(U) The radar tracking problem (RTP) is used as a medium to explain the concepts underlying BB programming. The RTP is particularly amenable to BB problem solving and has the potential to control radar resources as part of the solution. The BB database is constructed solely of flavors that act as data structures, as well as method-bearing objects. Flavor instances form the nodes and levels of the BB. The methods associated with these flavors form the basis of a distributed BB monitor and support the knowledge sources (KSs) in modifying the BB data. A rule-based system is used to construct the knowledge source activation record (KSAR) queue, and the goal nodes form the database. The prioritized KSAR queue solves the control problem associated with choosing the next KS. The BB is constructed in LISP with KSs in either C or LISP. The resulting program is used as a test bed for the RTP.

CONTENTS

Section 1. Introduction 3

Section 2. BB Architecture 6

Section 3. Radar Tracking Problem 8

Section 4. Representation Problem–Flavors 11

Section 5. Blackboard in Flavors 19

Section 6. Blackboard Knowledge Sources (KS) 28
- Hit Generation KS (GETBEAM) 30
- Assignment Problem 31
- Track Formation KS (GETTRACK) 34
- Spline Interpolation KS 34
- Segment Verify KS 36
- Merge Segments KS (MERGE-SEGMENTS) 38

Section 7. Blackboard Control 39

Section 8. Conclusions 52

References 61

Appendix 65

ACKNOWLEDGMENT

Seth Hutchinson's expertise in AI programming was invaluable in the preparation of this report. Discussions with him about design decisions provided a sounding board that resulted in a better product.

THE RADAR TRACKING BLACKBOARD (RTBB)

1
INTRODUCTION

Significant effort has been expended to solve the radar tracking problem (RTP), which is defined as finding the best partition of radar returns into disjoint time sequences that represent the trajectories of objects in the search area. Very sophisticated algorithms already exist to optimally map radar returns grouped by time into disjoint object trajectories [Reference 1]. The standard representation of these trajectories is the Kalman filter, which is an optimal model for tracking in the presence of Gaussian noise. Each trajectory is represented by a distinct Kalman filter. Once the radar returns have been assigned to distinct tracks, this information is abstracted, integrated, and evaluated to detect critical situations or make decisions. The RTP solution is the set of all trajectories and the inferences that can be drawn from this information. For example, certain trajectories represent threats to safety in an air traffic control system. So, any tracking system must keep accurate track files for all aircraft in the search area, and detect potentially hazardous situations early to warn the system users.

For large search spaces, distributed problem solving is an attractive alternative provided the search space can be partitioned in space, time, or both; for example, space could be partitioned by a division of azimuths and elevations. Distributed problem solving should allow the application of a blackboard (BB) to each partition separately. The final solution could then be synthesized by pooling the separate solutions; pooling the solutions is usually referred to as cooperative problem solving [Reference 2]. Therefore, the overall

method would be divide-and-conquer, and the BB approach would allow parallel processing to speed the solution at both local and global levels. Ultimately, the BB solution could also include control of the radar itself. For example, with an active array radar, where scanning may be dynamically controlled, control of the scanning pattern of the radar could also become part of the optimal solution, in the sense that the BB control problem could include control of the scanning pattern, leading to more efficient solutions to the RTP.

> *"The blackboard model is a relatively complex problem-solving model prescribing the organization of knowledge and data and the problem-solving behavior with the overall organization."* [Reference 3, p.39]

The BB paradigm is intuitively pictured as a number of scientists gathered around a BB and attempting to solve a difficult problem. Each scientist works on a part of the problem, but he or she may only communicate with other scientists by writing requests or results on the BB. Note that BB is an acronym for the blackboard system as well as the database associated with the system. The context should clarify the reference. Thus, the problem solving is coordinated only through the BB. This approach limits the degree of cooperation but minimizes communication protocols and misunderstandings. If scientists work on relatively disjoint parts of the problem, this approach will maximize the degree of parallelism.

The BB approach has been used in a number of successful systems, including Hearsay [Reference 4], HASP/SIAP [References 5 and 6], and PSEIKI [References 7 and 8]. These systems include applications such as general problem solving [Reference 9], planning [References 10 and 11], and image understanding [References 7, 8, and 12]. Image understanding BBs are usually hierarchical structures where the lower levels represent signals and/or preprocessed data, and the higher levels represent more symbolic information refined from the data [Reference 8]. As Nii [Reference 3] points out, the nodes on the BB represent partial solutions to the problem. The RTP has probably been solved using a BB, although this information is difficult to obtain [Reference 13].

Our RTBB is constructed in LISP with knowledge sources (KSs) written in either LISP or C. KSs solve various portions of the tracking problem. The main BB process is constructed in LISP, and the KSs

are either children of the main BB process or run under the BB process. The database, BB monitor, and scheduler are all part of the main BB process. All the processes run under the UNIX operating system. Each level and node on the BB is a flavor instantiation. These flavor instantiations are method-bearing data structures that are part of Franz LISP. A method is a procedure that is invoked by a message to a flavor instance. The method triggered depends on the message sent and the type of flavor receiving the message. For convenience, flavor instances are referred to as flavors, instances, or nodes interchangeably.

The methods associated with BB nodes act as local monitors collectively forming a distributed BB monitor, as scribes for the KSs in updating the BB information, or as information agents for the rule-based planner. After-methods that are written for the data nodes trigger after a node is altered and enter the changes on the goal BB. This implementation of the monitor using flavors is one of the more interesting aspects of the BB design. Reported changes are mapped to KS activations via a rule-based system whose database consists of the nodes of the goal BB. The KS activation records (KSARs) created by the rule-based system are prioritized, and the KSs are activated using the priorities of the KSARs. This BB is meant only as a test bed or a learning tool. A BB to run a real-time radar system would, of course, require much additional system-design effort in the real-time aspects of the problem.

Section 2 of this report gives a more detailed exposition of the model, and describes the purpose and interaction of all the components. Section 3 explains why the BB problem-solving method is a natural solution paradigm for the RTP. Because flavors are virtually the only data object in the BB, Section 4 describes those aspects of flavors that are particularly useful. Section 5 describes the BB system and database. Section 6 describes the KSs associated with the BB, and Section 7 describes the control flow and scheduling of the KSs. Section 8 contains the conclusions. Finally, the Appendix contains examples of how the problem-solving activity is carried out by the BB.

2
BLACKBOARD ARCHITECTURE

This section describes general aspects of the BB solution methodology and is based in large part on H. Penny Nii's knowledgeable tutorials on BB systems [References 3 and 4].

The BB architecture, which emerged from the HEARSAY-II effort, has been used by several systems [Reference 3, p. 38]. The BB is both a problem-solving paradigm and a reasoning and control architecture. By definition, a problem-solving model is "a scheme for organizing reasoning steps and domain knowledge to construct a solution to a problem" [Reference 3, p. 38]. A BB usually consists of three parts—the global database, the KSs, and the control. The global database is the BB and the only means of communication between the KSs. The KSs are procedures capable of modifying the objects on the BB and the only entities allowed to read or write on the BB. The control is the planner that selects the next best KS to be activated; that is, solves the control problem. This sequence of KS activations is called *opportunistic reasoning*, which supposedly advances the solution the most.

One view of a BB model is as a highly refined production system [Reference 10, p. 297]. In fact, both the model and system have the same three major components. A production system consists of a global database, a set of rules, and a control. The BB system database consists of all the data, and the partial and final solutions to the problem. This database is also called the BB. Corresponding to the rules are the BB KSs, which are generalized rules capable of both logical operations and numerical procedures. The KSs contain all the domain knowledge of the system. Corresponding to the control is the opportunistic reasoning, which attempts to choose the next KS to execute, based on the current state of the BB. Ideally, the KS chosen advances the solution the most [Reference 14].

There is a great difference between understanding the concept of a BB model and the implementation of a BB.

> *"The difficulty with this description of the blackboard model is that it only outlines the organizational*

> *principles. For those who want to build a blackboard system, the model does not specify how it is to be realized as a computational entity; that is, the blackboard model is a conceptual entity, not a computational specification. Given a problem to be solved, the blackboard model provides enough guidelines for sketching a solution, but a sketch is a long way from a working system."* [Reference 3, p. 29]

The BB control may be event, goal, or expectation driven. In the RTP, the BB initially was event driven and expanded to be goal driven. Events are changes to the BB, such as arrival or modification of data by the KSs. In an event-driven BB, the scheduler uses the events as the primary information source to schedule the KSs. The goal-driven model is more refined and uses a composite mapping from the events to goals, then from goals directly to KS activations, or indirectly from goals to subgoals. This refinement permits a more sophisticated planning algorithm to choose the next KS activation. Using goals, you can bias the BB, generate other goals to fetch, or generate other components of the solution [Reference 9]. If goals are isomorphic to the events, the BB is essentially event driven. With subgoaling, you can gain from the best of both event- and goal-driven BBs.

The efficiency of a BB implementation for solving a particular problem depends on how well the BB structure is tailored to the problem, and how the collection of processes flows through both the hardware and software. The performance analysis of BBs is a neglected problem whose importance will increase as the BB moves from research laboratories to the field [Reference 15]. Then a rush to develop the modeling skills and simulation support to characterize the BB response time and throughput will occur. This important issue is only mentioned here and will not be further addressed.

BB shells, which may be excellent research tools, are emerging. The generic BB (GBB) constructed by Corkill, et al. at the Department of Computer and Information Science, University of Massachusetts, Amherst, Mass. [Reference 16], and the Ariadne-1 BB system described by Criag [Reference 17] are two examples. The GBB is constructed by a group whose interest includes distributed problem solving, especially using BBs [References 9 and 16]. Thus, good BB shells probably will be available soon for exploratory problem

solving. However, it is doubtful you could build a BB in a real-time environment using a general BB shell.

Description of the general form of a BB architecture is based on the functional description given by Nii [Reference 3, pp. 43-44]. In this description, the KSs are procedures that advance the solution of the problem and record these advances by modifying the structures on the BB. The BB holds the state information as data objects in a hierarchical format, which the KSs reference and modify. The solutions are graphs whose vertices are the flavors or data objects; the edges are the relationships between the flavors. The control consists of the monitor and the modules that schedule the next KS to execute, i.e., the focus of attention.

3
RADAR TRACKING PROBLEM

The RTP is defined as finding the best partitiion of given radar returns into disjoint time sequences that represent the trajectories of craft or other moving objects. For aircraft flying in tight formations, we will associate a single trajectory with each formation. Each trajectory, whether associated with a single aircraft or a formation, will be called a track. Because aircraft may break away from a formation, any single track can lead to multiple tracks. The RTP consists of assigning a radar return to one of the existing tracks or allowing the radar return to initiate a new track. This problem is not new and has been solved with varying degrees of success and implemented on numerous systems. In fact, a BB solution of the RTP may already exist, although the solution is probably proprietary. The fact that TRICERO has an embedded radar tracker has been indicated [Reference 14, p. 96]. However, whether the tracker is implemented as a BB is not clear [Reference 13]. The RTP is particularly suited to the BB problem-solving method and is a good pedagogical tool. Moreover, the RTP is easily sized, which means that (1) at one end of the spectrum, a toy BB can be produced using this problem to test and explore BB concepts, and (2) at the other end, a lifetime could be spent constructing a distributed problem-solving system made up of BBs designed to control and solve the RTP in incredibly complex environments.

However, you may ask why you should use a BB model of a problem that presumably has been solved by other methods. The best of several reasons is that a BB allows a large degree of parallelism, which also fits well into a distributed problem-solving framework. Fortunately, criteria that can be used to judge the applicability of the BB model to the problem exist [Reference 4, pp 102-3].

The solution space consists of the data that are time-stamped radar returns, along with all the tracks and partial tracks. This information alone can create a large solution space, because the number of aircraft that can be included in a large search space around a modern airport might be very large. The radar returns vary widely in quality. Returns may have high S/N ratio in uncluttered backgrounds, but may also be noisy, cluttered, and weak. Obviously you design for the worst case, which includes noisy and unreliable data. Noise and clutter induce track anomalies, such as fades, splits, merges. Track formation in a noisy environment requires not only significant signal-processing procedures, but, in general, forward and backward reasoning at a symbolic level.

High-noise environments produce uncertain track information, and backward reasoning can verify track information via a hypothesis-and-test scheme. KSs used during this part of the reasoning may require higher spatial resolution and longer signal integration times to verify hypotheses. Expectation-, goal-, and model-driven reasoning are possible and desirable to best choose the next KS. So, a need to use multiple-reasoning methods exists. In addition to multiple-reasoning methods, the system must also reason simultaneously along multiple lines. For example, when track splits occur, watching and maintaining several alternative track solutions before modifying the track information on the BB may be desirable. Multiple lines of reasoning can play a natural role in searching for the optimal solution under these conditions. The last criterion requires that the current state of track information always be available, even though this information may be incomplete and uncertain. Returning an answer such as "Wait—system still processing" is unacceptable. Both pilots and controllers need to see the entire track solution evolve in real time.

The BB lends itself to sensor fusion. When infrared sensor and intelligence data are included with the radar data, the solution space is both large and diverse. Intelligence information introduces model-driven and expectation-driven aspects to the reasoning. So, a variety of input data and a need to integrate diverse information exist. The sensor data and the intelligence information may be independent and may cooperate in order to confirm a track. A consistent solution will require both independence and cooperation of the KSs.

Based on these criteria, the RTP is suited for a BB problem-solving paradigm, and can be expanded in numerous directions to address the complexities of actual systems. For signal-processing environments, the RTP is an archetypical example providing a good test of any problem-solving architecture. The RTP is also an excellent candidate for distributed problem solving [Reference 9]. The problem, which is naturally partitioned in space, can be cooperatively solved by multiple BBs working on each partition.

BBs can simplify software development. The BB system solves a problem subject to the constraint that the processes are independent enough to interact only through the BB database. This constraint may limit efficiency in achieving the solution, but also tends to maximize the amount of parallel processing that can be achieved. Another advantage of this design [Reference 4, p. 104], is independent development. In a multisensor environment, the information kept on the BB for tracking is in terms of coordinate vectors and their derivatives along with signal information. This information is not only process independent, but independent of the data structures used. Thus, parallel development and testing can take place in designing KSs because they are only coupled via the database and the scheduler.

However, a high price for the maximal separation of KSs via the BB database is overhead. For example, if no shared memory exists, the cost of data transfer between the BB and KSs can be very high in terms of real time, not to mention software design time. For research and development, this may be a small price to pay. But in real-time environments, this may not be acceptable. Also, the opportunistic control in a BB may be ideal from a conceptual viewpoint and may increase solution convergence; but, because opportunistic control is difficult to model mathematically, this control may lead to unpredictable behavior by the BB under circumstances not taken into

account during the test phase of the system. In spite of these drawbacks, BB systems inevitably will work their way into system designs.

4
REPRESENTATION PROBLEM—FLAVORS

The representation problem is central to problem solving in general. Implementation of a chosen representation requires suitable data structures. In a BB, each level can have its own representation, and consistency between levels is not necessary. However, in this implementation, flavors have been used to represent the data at each level of the BB and to implement the BB itself. Several reasons for this decision exist. Flavors are versatile data structures that are easily initialized and have built-in constructors, selectors, and mutators. In addition, flavors are method-bearing objects, and their methods can be used to monitor and update the BB itself. The following paragraphs expand on the properties of flavors, which have proved useful in this BB implementation.

Flavors are method-bearing objects with instance variables that may be easily redefined [Reference 18]. These instance variables, or just variables, may be instantiated to numerical values, symbolic values, lists, or symbolic expressions (s-expressions). Thus, the flavor composition is appropriately tailored to the abstraction level in the BB. In addition, very sophisticated ways of mixing flavors exist, i.e., constructing flavors built up of other flavors. We will extensively use before-methods and after-methods that can be attached with a flavor. The former types of methods initiate procedures before altering a flavor instantiation, and the latter types after. You may note that invoking methods in sequences more complex than implied by the names before-methods and after-methods is possible. Only a small portion of the flavors' potential has been tapped for this project. Some of the key properties of flavors are described in this section.

Flavor creation is achieved via the define flavor (defflavor) function, which defines the name and characteristics of s-expressions

in the structure. Consider a track node, the highest abstraction on the BB.

```
;;          tnode means track node -
;;       the flavor holding info on the track level

(defflavor tnode (
     (type 'track) ; the type is track
     (time 1234)   ; the last timestamp in the track
     last-coord    ; latest position of the track
     last-velocity ;  latest velocity of the track
     threat        ;   true if interval straddles zero
     snode         ;   backward pointer list to snode
     cpa-bracket   ; bracket about x and y
     check         ;   spline check of segment group
     checklyst )   ;   and list that must be checked to verify track
     () ; other flavors included must be placed inside this parentheses
     :gettable-instance-variables   ; allows send to ask for current value
     :settable-instance-variables   ; allows send to set the variables
     :inittable-instance-variables ; allows variables to be set at creation
)
```

The defflavor function is followed by a list of variables that may be initialized in more than one way. Following these variables is a place to mix in other flavors; finally, there are several options listed to apply to the variables. Flavor instances may be initialized via two simultaneous avenues. First, you can specify a default initialization for instances of a flavor during the defflavoring function. Second, you can initialize instance variables when the flavor instance is created; the latter may supplement or override the default initializations. In this example, the flavor called tnode contains eight variables. Two of these variables take default instantiations that will occur in every instance of the tnode flavor. The *time* variable will be automatically set to 1234 and the *type* will be set to 'track. However, this default initialization can be overridden at the creation of the instance by using the flavor option :inittable-instance-variable. Thus, for example, if the creation statement was

```
(setq 'track (make-instance tnode :time 2222 :threat 'true))
```

then the initial value of *time* would now be 2222 and the initial value of threat 'true. In the latter case, the unspecified default value of nil is changed to 'true; and, in the former case, the default value of *time*, 1234, is overridden to be 2222. In the creation of abstract data objects, data constructors, such as setq, are procedures that

make data objects [Reference 19, p. 98]. So, the flavors have a simple yet powerful set of constructors.

Flavors also have natural selectors and mutators built into their instantiations. Selectors extract information from data objects and mutators alter information in the data objects. Both of these mechanisms are embedded in the send operator, which sends the flavors (objects) messages to perform operations. Flavor operations are optional and are declared in our example by specifying the gettable and settable options in the instantiations. The selector gets information by sending a message to the object with the variable name. For example, (send track :time) will return the current instantiation of *time* in the tnode flavor instance called track. The mutator uses the same format, except now the variable name has :set- prepended to it; so (send track :set-threat 'false) alters *threat* variable of the track instantiation to 'false. The object-oriented nature of the flavors is a real advantage for obtaining and altering the contents of the BB nodes.

As discussed before, a most useful feature of flavors is that they are method-bearing objects, each method being invoked by sending a suitable message to an object. The operation in the message and the object combine to uniquely define the procedure that must be used to execute the method. Before- and after-methods are executed before and after specified operations, such as :set- or :init. Usually, these two types of methods are useful for massaging the data received and storing information in other instance variables. Because an after-method may be invoked after initializing or altering critical variables in a flavor instance, such specialized methods can report the changes to a queue or another portion of the BB. In an event-driven BB, the changes are reported to an event queue; and, in a goal-driven BB, the changes are reported to a buffer in the centralized monitor or directly to the goal side of the BB. The RTBB has been implemented as a goal-driven BB similar to that used by Lessor and Corkill [Reference 2], so the changes are reported directly to the goal panel or goal side of the BB.

As discussed in Section 1, before- and after-methods can be used to implement a distributed monitor whose job is to report changes in the BB database as goals. However, alternatives for designing monitors do exist. For example, polling techniques, along with change bits or variables in the flavor instantiations, could be

used to create a centralized monitor. As another alternative, KSs themselves could report all the changes to a centralized monitor, because KSs are the only entities allowed to alter the BB. In a way, you can think of before- and after-methods as being part of the KSs, or as a shared utility of these KSs for reporting changes, or further updating or altering the flavors.

The following defmethod is an example of a method that places a node on the goal BB after the *time* variable is set by an after-method.* The *time* variable is updated as new return information percolates up to the track level. In this new goal node, the variable *source* is set to the name of the flavor instantiation that invoked the method. The instantiation in this case is the internal identity of the tnode whose *time* change invoked the method. The variable *action* is set to 'change to reflect that the goal was caused by changing the time value, in contrast with, for example, a goal node created by subgoaling. The variable *type* takes the value 'track for obvious reasons and the variable *time* inherits the updated time value. The variable *threat* inherits its value from the tnode that invoked the method. The variable *snode* is a pointer to the snode that supports this track node.** Finally, the variable *duration* is instantiated to 'one-shot, so only one attempt is made for this goal node to be satisfied. Note that in the syntax of a defmethod, the symbolic name following each variable, such as :snode, is the name of a variable

---

* The reader already familiar with RTBB may be puzzled by this defmethod because it creates a track-level goal node from a change in the track level on the data panel. Usually, a track-level goal node is created by the addition of a segment on the data panel. The purpose of the goal is to merge the segment with one of the existing tracks or to start a new track with the segment. However, RTBB also needs facilities to create track-level goals directly from changes in the tracks because of the need for verification and possible subgoaling if the track is a threat, which means if the average velocity vector representing a track is aimed directly at the origin of the coordinate system. Verification consists of making sure that all the segments are similar in the polynomial sense, as discussed in Section 6. When a track fails verification, subgoals that check each segment against the average properties of the track must be created. If found to be too different, a segment must be released from the track and allowed to participate in the initiation of a new track. This defmethod could lead to formation of KSARs that could produce these subgoals.

** As will be explained in Section 5, radar returns, in the form of hits, are grouped into segments, which are then grouped into tracks. In RTBB, each segment is represented by a node called snode.

from the flavor to which the method is attached; if the symbolic name is quoted, the symbolic name is used directly.

```
;;
;; this defmethod pushs a node onto the goal blackboard
;;

(defmethod (tnode :after :set-time) (value)
     (sendpushgoal
     (make-instance    'bbevent
     :source    self
     :action    'change
     :type      'track
     :time      time
     :threat    threat
     :snode     snode
     :duration  'one-shot)
     tracks))
```

The flavor instance is placed on the goal side of the BB at the track level by the macro called sendpushgoal, which pushes a goal onto the track level using a send message. The sendpushgoal macro is a procedure that pushes an instance of the bbevent flavor onto the track level of the goal panel. The macro looks as follows:

```
;;  this macro pushes an object onto the level on the goal BB

(defmacro sendpushgoal (object level)
   `(send ,level :set-left
      (push ,object (send ,level :left))))
```

So, the set of goals on the track level of the goal BB is just a stack of these flavor instances. This method is invoked after a change has been made to the *time* variable of the track node on the data panel. This change occurs whenever the track node is updated, and the message that triggers this change looks something like (send tnodeptr :set-time (list newtime)). When only one or two methods are associated with each node type, writing one method for each variable is a simple matter. However, as the number of variables associated with each node on the BB increases, the coding of the methods becomes cumbersome. Seth Hutchinson, of Purdue University, suggested and wrote a macro to generate these methods automatically. The following version is a modification of that macro, which is designed for a goal-driven BB.*

* Seth Hutchinson, personal communication with Paul Kersten, 1987.

```
;;
;;
;;   This macro generates a flavor and the corresponding after-
;;   demons which report changes of a bnode, tnode, or snode
;;   to the goal panel.  Thus generating the defmethods forming
;;   the monitor
;;

(defmacro newflavor (flav level var-list var-sub inher-list &rest options)
   (cons 'progn
   (cons
   `(defflavor ,flav ,var-list ,inher-list ,@options)
   (do*
   (
   (worklyst var-sub (cdr worklyst))
   (op (car worklyst) (car worklyst))
   (mlyst nil)
   )
   ((null worklyst) (return mlyst))
(setq mlyst
(cons  `(defmethod (,flav :after ,(keywordize (concat :set- op)))
      (value)
   (sendpushgoal
      (make-instance 'bbevent
         :source self
         :action 'change
         :type type
         :variable ',op
         :coord coord
         :number number
         :time time
         :duration 'one-shot
         )
   ,level))
   mlyst)
)))))
;;
;;;
```

In this macro, a flavor is created of type flav with variables *var-list* and inheritance list inher-list, i.e., with all the variables and options normally available with any flavor. In addition, the variables contained in the var-sub list have update methods automatically generated by the macro code. If any of these variables is altered, the automatically constructed after-methods push goal nodes onto the proper levels of the goal panel.

To construct these methods, the macro generates a program that returns the list of methods created in the do* loop. When

finished, the macro executes the progn statement constructed, which includes creation of the flavor and associated methods that report changes to the goal panel. Note that the macro keywordize is a procedure used to intern the :set-op name into the keyword package so that the flavor features of Franz recognize this operation. An example of newflavor's use follows:

```
;;
;;snode is from segment node
;;the flavor holding info on the segment level
;;

(newflavor snode tracks (
        type    ;       is segment
        time    ;       this is the time of last coord
        coord   ;       note this is a coordinate list
        number  ;       number of points the the segment
        cpa     ;       closet point of approach a vector
        linear  ;       (position velocity)
        tnode   ;       ptr to a track node
           threat ;     true or false - updated  by tnode
        )
    (number)
()
:gettable-instance-variables
:settable-instance-variables
:inittable-instance-variables)
;;
```

Note that the data-segment node is set up so that when the number of points in the segment is changed, a goal node is pushed onto the goal side of the BB at the track level. To generate equivalent code without this macro, you first need to define a flavor using the defflavor function, and then add the following method.

```
(defmethod (snode :after :set-number) (value)
   (sendpushgoal
   (make-instance  'bbevent
      :source self
      :action 'change
      :type type
      :variable 'number
      :coord coord
      :number number
      :time time
      :duration 'one-shot)
   tracks))
```

Note that some of the variables, such as *source*, action, do not appear in the newflavor call, but do appear in the defmethod call. The definition of newflavor automatically sets these variables to certain fixed values. The newflavors macro is an illustration of the power of macros and the ease with which an impressive array of methods in a BB shell can be created.

At each level of the BB, the nodes are flavor instantiations. Each level is also a flavor instantiation. For example:

```
;; This is the flavor which makes up the
;; levels of the blackboard hierarchy

(defflavor bblevel (
        up   ;;        for higher level in BB hierarchy
        left ;;        for the goal BB panel
        right ;;       for the data BB panel
        down) ;;       for the lower level in BB hierarchy
()
:gettable-instance-variables
:settable-instance-variables
:inittable-instance-variables)
```

The following program statements create the segment level of the BB, and set the pointers held in the *up* and *down* variables to link the levels to one another.

```
(setq segments
   (make-instance 'bblevel :down nil :left nil :right nil))
;;;
(send segments :set-up tracks)    ;; links bottom level to top level
(send segments :set-down hits)    ;; links top level to bottom level
```

The *right* variable, which will be instantiated to a list of data-segment nodes is a data BB level. The *left* variable allows us to refer to the corresponding levels on the goal panel. Because the variables are allowed list instantiations, both the *right* and the *left* variables serve as storage sites for the data and goal nodes, respectively, at different levels of the BB—the segment level in the above example. In effect, the lists that become instantiations for the *right* and *left* variables act as queues or stacks of flavors, depending on their queueing discipline. This is convenient when you wish to apply some function on the entire set of nodes, because you may mapcar the function onto the list that is simply obtained via (send segments :right) message. Figure 1 illustrates the *left*, *right* organization of the BB; the right panel stores the data at different abstraction levels, and

the left panel stores goals, again at different abstractions, for the purpose of control. Further advantages of flavors will be more evident in the description of the BB itself.

Although RTBB is constructed entirely of flavors, the variables in the flavors may be instantiated to any s-expressions, such as lists. Any list may be used as a queue or a stack. We use the word queue in a generic sense and associate three components with it—arrival process, queueing discipline, and service mechanism [Reference 20]. The arrival process is characterized by an interarrival-time distribution for items stored in the queue. The service mechanism is composed of the servers and service-time distribution; note that multiple servers (e.g., processors) can cater to a queue. The queueing discipline describes how an item is selected from the queue. Stored items may be queued and waited on for service via some algorithm, or discarded completely. Therefore, we can use the same definition for LIFO, FIFO, or any generalized queueing system. When the type of queueing system is important to the discussion, the system's service discipline, like LIFO, FIFO, will be elaborated on.

5
BLACKBOARD IN FLAVORS

To design the BB system, define each of the three components—database, KSs, and control. The database, or BB, contains all the nodes that form the partial solutions to the RTP. Because all the data structures are flavors, specification of the hierarchical structure of the database and data structures should complete the BB database definition.

This BB consists of three abstraction levels and two BBs—a data and a goal BB. These sub-blackboards are called panels. The first-level data nodes are called beam nodes (bnodes) or hit nodes (hits) and are defined as follows:

```
;;  definition of hit node or beam node

(newflavor bnode nil (
    type    ;          type is hit
    time    ;          time stamp associated with coordinates
    coord   ;          list of the coordinates assoc with time
    number  ;          number points in the list
           )
           ()
    ()
    :gettable-instance-variables
    :settable-instance-variables
    :inittable-instance-variables)
```

Note that the newflavor macro was used to define the bnodes, although, in this case, no methods were automatically generated because fourth argument is an empty list. The second argument is set to nil for the same reason. *Coord* is a list of four-tuples corresponding to (t,x,y,z) coordinates of a radar return. The variable *time* is the time stamp of the return, which is the integer representing the number of time units since the system started. Unit time intervals are usually chosen to normalize the actual system parameters. The variable *number* is the actual number of distinct returns received at the time instance that corresponds to the time stamp. The node *type* is hit and specifies the abstraction level. For this BB, hit and beam nodes are treated the same. In practice, a beam of information is more primitive than a hit because the latter is a time-integrated sequence of beams. Hit nodes are generated every n^{th}-clock cycle where presently n is set to four.

A method may be used to refine the data before reporting changes. For example, before reporting the change to the goal BB, the following method first calculates the number of radar returns, enters that number in the bnode, and only then reports the change. Alternately, *number* could have been set directly. This method is a trivial illustration of the data modification capabilities of the monitor methods.

```
;;
;; an after-method which first updates the number
;; of returns and then reports to the goal panel
;;

(defmethod (bnode :after :init) (value)
   (setq number (length coord))
   (sendpushgoal
      (make-instance 'bbevent
         :source self
         :action 'change
         :type 'hit
         :variable 'coord
         :coord coord
         :number number
         :time time
         :duration 'one-shot
         )
      segments)
   )
```

This method would be triggered after the creation of a hit or beam node—for example, (make-instance 'bnode :coord coord). Here the variable *coord* is instantiated via the built-in methods specified by the inittable option in the flavor definition of bnode. Note that inclusion of ':init' in the first line of the defmethod ensures that the method would be executed on creation (or initiation) of an instance of bnode on the data panel, and on initialization of any of the variables in that bnode. The goal created by making an instance of the flavor bbevent represents the desire to extend existing segments using the data in the bnode.

Interestingly, this method alters the bnode whose creation causes the execution of the method. The bnode is altered because the variable *number* now has an instantiation equal to the length of *coord*. This may seem at variance with the point made in Section 2. In that section we said that in an ideal conceptualization of a BB architecture, only KSs should be allowed to alter information in the BB database. What we have accomplished with the method just described is not at variance from the ideal. That aspect of the defmethod, which updated the value of *number*, could have been incorporated in the KS that created the bnode in the first place. You can view this data-refinement aspect of methods as extensions of the KSs or as some distributed KS. One advantage is that such methods simplify the coding of interfaces between the BB process and the KSs.

The next level of abstraction on the data panel is the segment node (snode). Segments are defined for convenience and represent a small number of hits (a fixed number chosen by the designer), which can be adequately modeled as a linear segment. By fitting linear segments to the returns, we reduce the sensitivity of the system to noise spikes. Segments approximately colinear are grouped together to form tracks. Tracks will be discussed later in this report. A track will not be started unless a segment is longer than a certain minimum number of points, usually two. In addition, the most recent hit in a segment older than 10 time units is automatically purged from the BB database. If a track consisted of only one segment that was purged because of the time-recency requirement, the track would also be purged. Segment nodes are defined as follows:

```
(newflavor snode tracks (
    type    ;          is segment
    time    ;          this is the time of last coord
    coord   ;          note this is a coordinate list
    number  ;          number of points the the segment
    cpa     ;          closet point of approach a vector
    linear  ;          (position velocity)
    tnode   ;          ptr to a track node
    threat  ;          true or false - updated  by tnode
       )
    (number);          the variables that trigger a report
    ()
    :gettable-instance-variables
    :settable-instance-variables
    :inittable-instance-variables)
```

Note that the variable *coord* is a list of coordinates associated with a given segment and not with a given time instance, as in the beam nodes. That is, the coordinates are grouped via spatial continuity via temporal continuity as in bnodes. The variable *time* refers to the sequence of times corresponding to the coordinate points. So, both *time* and *coord* are stacks implemented as lists. The closest point of approach (*cpa*) is calculated by an after-method via position and velocity information contained in *linear*. The variable *linear* is instantiated to a list that consists of the position and velocity computed from the two most recent hits in the segment. Note also that the variable *cpa* is instantiated to the perpendicular distance from the origin to a straight line that is an extension of the two most recent hits in the segment. The variable *threat* is true if the instantiation of *cpa* falls within a small region around the origin,

otherwise the *threat* is false. The extent of this region is ε times the *last-coord*, and the comparison threshold is dependent on the distance because more distant craft have greater directional uncertainty. (This point will be explained further in the discussion on the GETTRACK KS.) Computation of a value for *cpa* for a given segment occurs when the segment node is initiated, so determination of whether *threat* is true or false does not occur until a track-level node is updated with the segment.

The highest data abstraction consists of track nodes. A track node is the grouping of approximately collinear segments. Two segments belong to the same track, if the following two conditions are satisfied. First, we must have $\cos^{-1}\theta > 0.9$, where θ is the angle between the velocity vectors for the two segments. The velocity vectors are contained in the instantiation of the variable *linear* for the segment nodes. Second, the faster of the two aircraft must be able to reach the other in one unit time. The second condition is necessary because we do not wish to group together segments for aircraft flying widely separated parallel trajectories. In general, only a single track node will exist for a single formation of aircraft, no matter how large the formation. If a formation splits into two or more formations, the original track splits into as many tracks. The track nodes are defined as follows:

```
;;
;;  tnode is of from track node -- the flavor holding info on the track level
;;

(defflavor tnode (
          type          ; the type is track
        time        ;    the last timestamp making the track
        last-coord    ;  latest position of the track
        last-velocity ;   latest velocity of the track
        threat        ;   interval straddles zero
        snode         ;   backward pointer list to segment node
        cpa-bracket     ; bracket about x and y
        check         ;   spline check of segment group
        checklyst     ;  and list for track verify and break
        )
   ()
   :gettable-instance-variables
   :settable-instance-variables
   :inittable-instance-variables)
```

The variable *type* is always instantiated to the atom track. The variable *time* is instantiated to the time stamp of the most recent hit

in any of the segments composing the track. The variable *snode* is a list of pointers to the segment-level nodes supporting the track. The variables *last-coord* and *last-velocity* are the latest average position and velocity vector associated with the track; averaging is performed by taking a mean of the position and velocity vectors associated with all the segments in the track. (The position and velocity vectors for each segment are contained in the instantiation of *linear*.) The variable *threat* is instantiated to t (for true) through an after-method by taking a disjunction of the *threat* values of all the segments in the track. The variable *cpa-bracket* is equal to the intervals along x and y, each interval being the union the *cpa* intervals associate with the segments in the track. If *threat* is set to t, a goal node is deposited at the track level whose job is to conduct a spline check of each segment in the track to confirm that the grouping of segments is coherent. Coherence is measured by the similarity of polynomial coefficients associated with fitting splines to the segments; this work is done by GETSPLINE KS. If the grouping of the segments is coherent, the variable *check* in the tnode in the database is set to t; if not, *check* is set to 'fail. Setting *check* to 'fail causes the formation of another track-level goal node at the next update of the tnode. This goal node is recognized by the rule-based planner, which deposits many subgoals for alternative grouping of the segments into possibly multiple tracks.

The abstractions for the goal nodes are identical to the abstractions for the data nodes, as shown in Figure 1. Nodes on the goal panel are used by the rule-based planner and the scheduler to focus attention of the BB. The goal nodes are used to create knowledge source activation records (KSARs), which are placed in a priority queueing system. The KSARs and the KSAR queue are flavors themselves.

The goal nodes are defined as flavor instances built up as mixtures of two flavors. The main flavor bbevent is mixed with the flavor goal-attributes, which contains duration and position attributes for the goal nodes. Duration refers to the length of time the goal is allowed to stay on the BB. For example, a one-shot duration means there is only one opportunity for the planner to test a node against the rules to see if the node matches any of the antecedents; if the match fails, the goal node is discarded. Most goal nodes are of one-shot type; for example, the goal to update a tnode with new segments. Only one KSAR for this goal node, which

contains a pointer to the segment used for updating, will ever be formed by the rule-based planner. The goal node is purged as soon as the KSAR is formed. Therefore, if this KSAR fails to satisfy the goal node, the goal node will not be there to re-attempt updating of the tnode with the same segment.

In addition to the one-shot type, RTBB also contains a recurrent goal node. A recurrent goal node is disabled after it satisfies the antecedent of specific rules, and is re-enabled after a KS is fired from the subsequently generated KSAR. Recurrent goal nodes are never removed from the BB, so they act much like synapses that have a latency period before they may be fired again. The job of the recurrent goal node currently in RTBB is to locate old segments, which are segments whose most recent returns are between 3- and 10-time-units old, and attempt to join these segments with more recent segments. Suppose the database at the segment level contains a snode composed of the following bnodes ($b1_1$, ..., $b1_n$), and the time stamp of $b1_1$ is 5, of $b1_2$ 6, and so on. Also, assume another snode exists that is made up of ($b2_1$, $b2_2$, $b2_3$) where the time stamp of $b2_3$ is 3. Then the job of the recurrent goal node is to merge the two segments, because the time stamp of $b2_3$ is so close to that of $b1_1$. The actual merging, carried out by the MERGE-SEGMENTS KS, will only take place if the extension of the b2 segment to the time instant corresponding to the beginning of the b1 segment is within an acceptable circle.*

The goal nodes at all three levels are created by making instances of the following bbevent flavor mixed with the goal-attribute flavor. Note the important distinction between the data and the goal panels. On the data side is a separate flavor for each abstraction level, but on the goal side a single flavor is used. The reason for this difference is that the goal nodes at all the levels form a database for the rule-based planner; therefore, their similarity is a convenience.

---

* In this explanation, a snode was shown as a list of bnodes. In actual practice, a snode is a list of coordinates and associated time stamps of the bnodes that form the snode. The actual bnodes are discarded as soon as these returns are assigned to prevent them from overwhelming the BB database.

```
;;  The goal node flavor called bbevent which is the basic
;;  goal blackboard node.
(defflavor bbevent (
        source  ;     generating  node
        action   ;    level this event affects
        type     ;    hit or track etc
        variable  ;   this is variable triggering event
        time      ;   may be list or number
        coord     ;   list of coordinates
        number    ;     number of coordinates
        threat    ;   for tnodes
        snode     ;   pointers to snodes
        pattern   ;   this is list used for pattern match
        )
    (goal-attributes)              ; mixed in flavor
    :gettable-instance-variables
    :settable-instance-variables
    :inittable-instance-variables)


;;
;;  The mixed in flavor representing the goal node attributes.
;;
(defflavor
    goal-attributes  (
        duration   ;  time latency of the goal node
        position   ;  position relative to coord
        goalptr    ;  pointer to other goal nodes
        conditions ;  preconditions to fire
        ksarptr    ;  pointer to ksar which is queued
        )
()
:gettable-instance-variables
:settable-instance-variables
:inittable-instance-variables)
```

For those goal nodes created by after-methods executed in response to new entries on the data panel, the variable *source* is instantiated to the internal identity of the data-panel node. On the other hand, when a goal node is created by the subgoaling process, *source* is instantiated to the internal identity of the tnode that caused subgoaling to take place. The variable *action* is usually instantiated to 'change, as can be seen in the definition of newflavor, to reflect the fact that a goal node was created by a change in the data panel. The variable *type* is set to the level at which the goal node is created, meaning that *type* is instantiated to either hit, segment, or track. The variable *variable* is instantiated to the name of the variable for which newflavor creates an after-method for reporting to the goal panel; this can be seen in the definition of newflavor in Section 4. The variable *time* is the time stamp of the data-panel node that

triggered the formation of the goal node. If a goal node is initiated by an snode, then *time* would be instantiated to a list of time stamps of the hits constituting the snode. If a track-level goal node is initiated by a tnode, then *time* is set to a single value that is the latest time stamp associated with the track. For a goal node at the segment level, the variable *coord* is instantiated to the list of coordinates of the hits to be assigned to segments. When a track-level goal node is launched by a tnode, then *coord* is left uninstantiated. For segment-level goal nodes, *number* is set to the number of hits in the radar return yet to be assigned; for track-level goal nodes, *number* is left uninstantiated. The instantiation for *threat* takes place by mechanisms explained earlier; basically, this variable would be set to t or nil. The variable *pattern* is not used at this time.

In the flavor goal-attributes, the variable *duration* indicates whether the goal node is one-shot or recurrent; flavor instantiations are therefore 'one-shot or 'recurrent. The next three variables are not being used at this time but have been included for possible future use. The variable *ksarptr* is left uninstantiated for one-shot goals, but is instantiated for the recurrent goal to the internal identity of the KSAR generated by the goal node. While *ksarptr* maintains this instantiation, the recurrent goal node is inhibited from launching another KSAR. The instantiation of *ksarptr* is reset to nil by termination of the execution of MERGE-SEGMENT KS.

Now that you are familiar with the organization of RTBB, we will reiterate, hopefully in a more precise manner, the overall method for solution formation. All the radar returns or hits generated on a scan of the search space are given the same time stamp. The list of hits occurring in one scan are contained in a flavor on the hit level of the data panel shown in Figure 1. A new list of hits triggers the distributed monitor to place a goal node on the segment level of the goal panel. This goal node represents a desire or request to use the new list of hits to update existing segments. If no existing segment can be found to match a particular hit, a new segment is started with the new hit.

The segment nodes on the data panel are supported by the hit nodes. The segment nodes are, in turn, grouped into track nodes. To drive the segment nodes to a higher abstraction level, or to push segments into tracks, you need to express this desire by establishing

goal nodes at the track level of the goal panel. These goals point to segment nodes that need to extend existing tracks or establish new tracks. Tracks are not established from segments unless the segments are at least 2 points long. (Any length threshold may be chosen because this is a constant parameter). A track may be thought of as an extended segment with the segments providing some buffering against spurious noise, which results in false segment starts. However, a track is more than an extended segment. A track may represent many segments, so that if several aircraft are in tight formation, these aircraft would be represented as one track, with the track characterized by an average position and velocity vector.

To process a track goal, a KSAR is generated via the nodes on the track level of the goal panel. The KSAR generation is accomplished via the rule base when rule 3 is fired. This rule requires that the node be of type 'segment, have more than one data point, and have an action variable instantiated to 'change. If all these antecedents are satisfied, then the create-ksar function is called, and a KSAR is created to extend a segment into a track or extend and update an existing track. The function create-ksar uses the information in the goal node to select the correct flavor instantiation for the KSAR. Figure 2 summarizes both the BB nodes and the KSs.

In general, a goal can only be achieved by activating a KS via a KSAR. So, a goal node must activate a KS directly via an appropriate KSAR or indirectly through subgoals generated from the goal. The priority of the KSAR generated by a goal node will determine the position of the KSAR in the KSAR queue. This queue is a complicated cyclic-priority queueing system, which determines the order in which the KSs are activated. However, in general, the higher the data abstraction on the BB, the higher the priority.

6
BLACKBOARD KNOWLEDGE SOURCES

Six KSs are part of the goal-driven BB. Each KS is a specialist solving a small portion of the problem, and each concentrates on a BB

object. The following is a list of these KSs and a short description of their purpose.

1. Hit Generation (GETBEAM)—This KS is written in C and simulates the trajectories for various aircraft.

2. Assignment (GETASSIGNMENT)—This KS assigns radar returns grouped by time stamp to returns grouped by proximity in coordinate space (a mapping from a time grouping to a spacial grouping).

3. Track Formation (GETTRACK)—This KS groups segments or linear fits by average trajectory, i.e., segments that are close in both coordinate and velocity space.

4. Spline Formation (GETSPLINE)—This KS checks the track groupings by testing to see if the trajectories are close in a polynomial representation space defined by the splines. To do this testing the BB must chain down the solution tree to construct the spline coefficients.

5. Verify Tracks (VERIFY)—This KS is used to verify a track that still matches a particular segment after the spline-formation KS has failed a track, indicating the segments are no longer consistent.

6. Merge Segments (MERGE-SEGMENTS)—This KS detects moderate-length gaps in the trajectory data, and then attempts to extend the older segments to the appropriate current segments. The KS attempts to match faded signal segments with newly emerging tracks, creating a longer and more established track.

Overall, the hit generation KS drives the BB with radar return samples. The assignment KS maps these samples into linear approximations of trajectories, and the track formation KS further groups these linear segments. The spline formation KS checks that the final trajectory grouping makes sense. The verify track KS breaks out tracks that fail the spline-grouping test. The BB will reform the track groups later. The spline formation and verify track KSs constitute a backward type of reasoning. Lastly, the extend segments KS attempts to maintain track continuity of weak or fading trajectories. These KSs are explained in further detail in the following paragraphs.

HIT GENERATION KS (GETBEAM)

This KS, written in C, uses Pratt and Faux's version [Reference 21] of Bezier's curves to determine trajectories for the hypothetical aircraft. In Bezier's curves, the trajectory is determined by a trapezoid formed from four vectors. These vectors are used to form a 4x3 matrix, which stores the vectors for different paths. Every time the program is initiated—when the BB KSAR runs GETBEAM—the program generates a set of coordinates, one for each trajectory. This program is easily generalized to include an arbitrary number of objects. In fact, several other versions are used to drive the BB under various scenarios.

The equation used by GETBEAM to generate the Bezier UNISURF curves is a single-vector equation that is a cubic in the parameter u. This real parameter u is a normalized time parameter, defined so that $0 < u < 1$. The equation is given by

$$r(u) = (1 - u)^3 r_0 + 3u(1 - u)^2 r_1 + 3u^2(1-u)_2 r + u^3 r_3$$

The generation of the trajectories occurs in another process, which runs the C program called testpath.c. This program is compiled, and its executable file is used via the *process command to interact with the BB. Whenever the n^{th}-clock event occurs (currently $n = 4$), a goal node is pushed onto the hits level of the goal panel. A KSAR, formed directly from this goal, activates this process using a command (GETBEAM), and the C program generates the next trajectory point. The step size of the trajectory is controlled via the step size of u, which is stored as a constant within the C program. Thus, for each set of aircraft trajectories, you need another C program to drive the BB system. Presently, several versions of these trajectories have been created with two or three targets to test various scenarios.

In an actual application, data would be buffered and the KS would probably handle batches of radar returns. From a queueing systems point-of-view, the returns represent arrivals from an infinite source. In this system, you have finite sources because the data are <u>not</u> allowed to enter the system until a command is issued to fire the data source. The difference is recognized but ignored because modeling of data sources is a secondary consideration. That

is, the internal data flow, not the arrival process, is the interesting process.

ASSIGNMENT PROBLEM

This KS takes the coordinates given by the hit nodes and uses these new radar returns to extend tracks. Thus, the KS solves the assignment problem by taking a fixed number of returns, say m, and matching them to n-fixed segments. The returns for a given time stamp are stored in hit-flavor instantiations on the hit level, and elementary tracks called segments are stored as segment-flavor instantiations on the segment level. Segments may be thought of as localized linear approximations to longer tracks. The segments are used to extend tracks, which are groups of segments. In practice, updates would be accomplished using a Kalman filter.

In this KS, the branch-and-bound procedure is used to solve the assignment problem. The branch-and-bound is in many ways the same as the best-first search procedure. To solve the problem, you must first limit the scope of the problem. The assignment problem is an essential part of a tracking problem. Samples are received from scans of the radar antenna system and are assumed to be received at the same instant. These samples must do one of the following:

1. Extend an existing track
2. Start a new track
3. Split an existing track
4. Merge two existing tracks
5. Terminate an existing track

The first two cases are handled by the GETBEAM KS. Cases three and four are handled by separate KSs, and case five is handled directly by the rule-base planner. All these cases are handled by the BB, except case two. The general problem is very difficult because of all the subcases. The basic cases are dependent on the relative number of data samples and established tracks, i.e., more data samples, the number of data samples, or fewer data samples than established tracks. When the same number of data samples and tracks exist, the assignment problem is the classic problem of assigning a number of jobs to the same number of employees. When more data points than

tracks exist, the unmatched data point must start a new track. And when more tracks than data points exist, one of the tracks is not updated. All of these cases are handled.

The problem is best restated by using graph theory when the number of segments and columns are the same. Start with a bipartite graph with n vertices in one partition of the vertices, say X, and m vertices in the other partition, say Y. Seek to pair the vertices in X to vertices in Y. That is, establish a perfect match in the bipartite graph. The cost of connecting vertex i of X with vertex j of Y is denoted C(i,j), and this matrix of costs forms the starting point of the branch-and-bound problem. The costs contained in C, for our case, are the Euclidean distance between the sample coordinate and the established track because the X vertices are associated with the new sample coordinates, and the Y vertices are associated with the established tracks. In terms of matrices, the problem can be restated as choosing one row for each column without repeating a row or a column. So for each row chosen, cross out that row and the column, and then do the same for each submatrix that results.

Figure 3 illustrates the complete bipartite graph for four hits and four segments. Note that the branch-and-bound algorithm avoids a breadth first search or total enumeration of all the paths in this graph, so the exponential complexity is avoided in most practical problems. The heuristic used is the distance or cost between the points. Figure 3a illustrates the typical case if the paths used the Kalman Filter confidence regions for the established tracts, and used highest probability as a measure to determine track matches when 90% ellipsoids overlapped.

The branch-and-bound problem can best be described when the number of solutions is countable—as in the case where the number of possible solutions is $n!/(n - m)!$, $n > m$. That you are trying to minimize a functional on the space is assumed. Branching corresponds to partitioning the solution space into subspaces for which a known lower bound can be calculated. This lower bound need not be a feasible solution. An upper bound on the solution is needed as well, and any feasible solution will serve an upper bound. The branch-and-bound procedure uses these bounds to truncate the search procedure for the solution. If the lower bound on any subspace exceeds the upper bound, the solution of the minimization problem is not in that subset. So, branch or subspace is eliminated

from the possible solution space. Partitioning or branching is continued until an optimal solution is achieved. The upper bound can be lowered at any stage, if lower feasible solutions are found enroute to the optimal solution. The branch-and-bound procedure used in this BB is found in Reference 22.

A method of implementing this procedure is the best-first search for the assignment problem. In this case, the branching is the same as expanding each node using its children. The evaluation function is $f = g + h$, where $g = g^* =$ the cumulative distance in the assignments between the established paths and the corresponding samples associated with those established tracks. The heuristic h is an admissible heuristic that is formed by summing the minimum of each column of the remaining submatrix constructed from the partial solution of assignments made so far. This value h is a lower bound on the actual distance and, in fact, may not be feasible. Moreover, the heuristic obeys the monotone restriction. If m is any child of n, $h(n) - h(m)$ is the minimum of the first column in the resulting submatrix created by choosing n as the last vertex of the partial path. Because C(n,m) is a member of that first column, then $C(n,m) \geq h(n) - h(m)$ and is the only requirement for the heuristic to obey the monotone restriction.

The expansion of the children at each node is the difference of the set 1, 2 , ..., n and the vertices making up the partial solution. With this latter definition, the branch-and-bound problem is seen to be an A* search, which is a special case of the graphsearch procedure described by Nilsson [Reference 23]. To speed truncation of the search tree at each node expansion, a simple, feasible path extension of the partial solution is made to see if the upper bound can be lowered. If f, for a given node, exceeds this upper bound, the graph is pruned at this node.

The best-first procedure, which is developed by Winston in his LISP book [Reference 19], forms the basis for this procedure with the appropriate modifications to truncate the search using the upper bound and revision of the upper bound at each node expansion.

TRACK FORMATION KS (GETTRACK)

This program takes the segments, or local linear fits, groups them, and represents the group by an average trajectory, provided the tracks are close in coordinate and velocity space. Close in coordinate space means within one time unit of travel for the fastest aircraft. That is, if the fastest aircraft turned directly toward the other aircraft, would the fastest aircraft intersect the other aircraft within one time unit. The velocity vectors are close if they are parallel or nearly so (i.e., the cosine of the angle is greater than 0.9). Other conditions may be added to ensure that the velocity vectors are more similar. This KS, called GETTRACK, is written in LISP and compiled using the Liszt compiler.

The track KS evaluates the threat of a track to the region near the origin. This KS may be thought of as a threat-assessment algorithm that, for example, calculates the threat to the airport traffic pattern. The two quantities needed for this algorithm are the current position and the *cpa* of the aircraft, both variables defined in tnode flavor. An error vector is formed from the difference between the *cpa* and the current position, that is, S(cpa - r) where S = 0.1. This error vector allows a confidence region to be formed at the origin for the *cpa* of the aircraft. If that confidence region includes the origin, then the aircraft is a threat.

The KS returns the threat assessment and the confidence region, which is stored in the threat flavor instance. Actually, placing more of the entire updating of the threat nodes in the KS, which is external to the BB process, would be better. To do this, however, would require passing a large number of variables or passing entire flavors. Moreover, the KS would have to access more information on the BB itself. In short, the interprocess communication interface would need upgrading, and the control loop would need a finer grain so that other processes could be executed while this information was passed back and forth.

SPLINE INTERPOLATION KS

This program, based on a spline routine in Reference 24, obtains a polynomial expression for the track between sample points based on the coordinates and time stamps held in the segment nodes.

The coefficients of the polynomial may then be compared to determine the fit of the tracks to each other. This KS is used in a hypothesize-and-test reasoning method to verify the track grouping of segments used in the track nodes.

The spline interpolation KS in this BB is a C program, which is designed to obtain a cubic fit to the trajectory data. The sample trajectories are generated by another KS and are obviously known. However, the point of this program is to assume that the analytic form of the trajectories are not known. To test that two aircraft are in formation, obtain a spline fit for the trajectories, and compare the coefficients of the polynomial fit represented by the splines. You are testing to see if the segments are close in a polynomial representation space. Although you may question whether this is the optimal space to measure the closeness of the shapes of trajectories, that doubt is not the point of constructing another representation.

The purpose of the splines is to test a reasoning method other than forward chaining. The spline routine is used in a hypothesize-and-test reasoning step, which in this form is not used to advance the solution, but verify that the conclusions are still valid by checking. The hypothesis is that a formation has been detected. In practice, the detection of a bundle of trajectories is a far more complex problem than modeled in this BB. To avoid getting sidetracked, detailed modeling of detection and estimation programs are ignored.

The spline program is based on the version given in Reference 24, which is in FORTRAN and has been converted to C. These are cubic splines, and the following discussion of the splines is based on Reference 24, Chapter 4. For a three-dimensional curve, we used three one-dimensional cubic splines.

The spline used here is based on the parameter u, which is the parameter $u \in S[0,1]$. What is nice about this approach [Reference 24] is that the authors define the parameters so that the spline s(u) is expressed as the sum of a linear interpolation and a cubic correction. In particular, define $h_i = u_{i+1} - u_i$ and w as the relative displacement between the u_{i+1} and u_i, that is, $w = (u - u_i)/h_i$ and $\bar{w} = 1 - w$. Now, the cubic spline is represented by the equation

$$s(u) = wy_{i+1} + \bar{w}y_i + h^2_i[(w^3 - w)\sigma_{i+1} + (\bar{w}^3 - \bar{w})\sigma_i]$$

where y_i and y_{i+1} are the function values for u_i and u_{i+1}, respectively. Notice that the last term is a third-order correction to the linear interpolation, and this correction is zero at both end points of the interval $[u_i, u_{i+1}]$, respectively.

The spline used in this system is a natural spline, meaning that $s''(u_0) = s''(u_{n-1}) = 0$ and the continuity conditions at the knots yield a set of simultaneous equations that are tridiagonal. Solving the simultaneous equations yields the coefficients in the cubic fit given by

$$s(u) = y_i + b_i(u - u_i) + c_i(u - u_i)^2 + d_i(u - u_i)^3$$

$$u_i \leq u \leq u_{i+1} \quad \text{for} \quad i = 0, \ldots, n - 1.$$

These coefficients are then compared to test the cubic fit of the trajectories of these aircraft. Thus, these coefficients will test not only position and velocity but also the higher-order path coefficients. This fit uses the sum of the absolute errors in these coefficients to test the null hypothesis that the trajectories are the same, i.e., the aircraft are in formation. This problem is singular because there is no noise in the system. A statistical version of this problem may be far more complex, because the statistics of the coefficients may not be easily derivable in terms of the statistics of the radar returns.

SEGMENT VERIFY KS

This KS, which is part of the BB process, merely examines each segment composing the current track to determine if the initial formation condition is satisfied. The examination is done by subgoaling. One subgoal is generated for each segment node by the rule base, then the BB checks each of these goals to verify that the segment is still within the track. If the track does not pass the verification test, the pointers to the segment from the track and vice versa are removed, and the BB reforms the tracks at a later time.

The test conditions are the same conditions needed to form the tracks in the first place, except that you are not comparing segment to segment but segment to track. The current position must be

within 1-second travel of the maximum velocity aircraft. In addition, the angle θ between the trajectories must have cosine$^{-1}\ \theta >$ 0.9. The test is basically an AND tree, which means the track will not be totally verified until all the children have been verified. During this verification period, the spline back-tracking algorithm, which initially detected the improper grouping of segments, is suspended. This suspension is accomplished by marking the track node *check* variable as failed, and having the spline goal node check that condition before the spline KS is fired.

Subgoaling is accomplished directly from the rules. In particular, the rule looks as follows:

```
;;
;;  This rule generates the subgoals needed to check tracks
;;
   (setq  rule2a
      '(rule   spline-check-failed-generate-subgoals
      (if
      (and
      (equal (send gnode :type) 'track)
      (equal (send gnode :threat) t)
      (equal (send (send gnode :source) :check) 'failed)))
      ;; ---------- generate subgoals --------------------
      (then
      (progn
      (create-subgoals-to-break-track  (send  gnode  :source))
      (format t "~% 2A 2a 2a 2a 2a 2a 2a 2a 2a FIRED ")
(format t "~% $$$$ rule spline-check-failed==> generate subgoals $$$$")
)
   )))
```

If a track node fails the spline check and the track is a threat, then subgoals are created. These subgoals are placed on the goal panel and mapped into KSARs with a high priority. During the verification of segments to this track, the segments are not prevented from being updated. The verification test for the track and segments is always updated to a common time so that time differences are properly normalized.

To implement the AND feature of the track node, a separate variable called *checklyst* is kept in the track node. If a track node fails its spline test, then the segment-node list is copied into the *checklyst*. As each verify-track KS is run, the last step is to remove the segment-node pointer from this *checklyst*. If this KS fails to reverify that the segment node should be part of the track, the KS

also removes the pointer from the segment-node list. An after-method is used to update the status of the track node only when the *checklyst* is changed back to nil. At that point, if the segment-node list is empty, the track node is removed from the data panel because the track node has no supporting hypotheses or segments. Otherwise, the *check* variable is reset to nil so that the spline back-chaining track check is enabled again.

This KS is written in LISP and is part of the BB process that avoids the transmittal of flavors across process boundary. The important point of this KS is that it breaks established tracks and uses subgoaling.

MERGE SEGMENTS KS (MERGE-SEGMENTS)

When radar signals fade, a segment and track will atrophy and eventually be removed from the BB. If the track reappears later, the track will be started as a new segment and then as a new track. The time lapse between disappearance and reappearance determines how the BB handles the problem. If the fade is sufficiently long, the original track is purged from the BB and the reappearing trajectory is handled as a new track. Between these two extremes the MERGE-SEGMENTS KS attempts to match given segments with established tracks.

The merge-segment algorithm is not commutative, because the algorithm extends the segments in a prescribed order and does not check all possible combinations of atrophied segments with established segments. The time window used as a precondition to this KS looks at the time since the last update. If this time is less than 10 but greater than three time units, a merge of the two segments is attempted. If a segment is eligible, the segment is extended in time and space, and its predicted position is matched against the established tracks to determine if the segment was the parent of the existing segment. The match is probabilistic in nature. A match occurs if the extended segment is inside the 0.67 percentile of the start point of the second segment. The confidence region uses an exponential distribution on the absolute difference of the position vectors $| x - y |$ with a mean of one-half the velocity magnitude. Intuitively, the mean is half the distance traveled in one time unit as predicted by the linear model of the faded segment. Use the tail of

the exponential distribution to form the confidence region. If $D = | x - y |$ is greater than a threshold c, then the match is rejected. The threshold is determined by the condition that $P(D > c) = 1/3$, where $P(X > d) = \exp(-d/\text{mean})$.

The MERGE-SEGMENTS KS is implemented within the BB process itself because the KS requires extensive access to the data nodes on the BB. If shared memory was available on the BB, you could implement the KS as a separate process. However, data transfer across the BB/KS interface is too great. Access to all the segment nodes is essential because there are several inquiries for each established segment node to implement the best match for the faded segment.

The goal-driven BB uses a recurrent goal node to monitor and schedule the MERGE-SEGMENTS KS. The MERGE-SEGMENTS goal is permanently placed on the goal BB. When segments satisfy the rule to activate the MERGE-SEGMENTS KS, a pointer in the KS goal is established to the generated KSAR. This pointer also acts as a flag inhibiting any further activation of the KS because the pointer's presence is checked by the rule base. Once the KS is activated and completed, the flag is removed and the rule base can satisfy the goal again. The name recurrent is derived from the definition of an alternating recurrent process in Stochastic processes. The on-off history of a light switch is an example of an alternating recurrent process. This implementation was chosen because it tends to run the MERGE-SEGMENTS KS on a continuing basis at a low priority and illustrates the flexibility of a goal-driven BB as both data and expectation driven. Example 4 in the Appendix demonstrates how the MERGE-SEGMENTS KS can handle short signal fades.

7
BLACKBOARD CONTROL

Control of the BB is opportunistic in nature, i.e., chooses the KS that most advances the solution. However, design of the optimum choice is ultimately the product of the programmer, who presumably has an understanding of the application domain. Each of these events is mapped into goal nodes, which in turn are mapped into

subgoals or activation records, and these structures are queued in the KSAR queue. The KSAR queue is a priority queue, and the priorities are the mechanism employed to choose the next KS activation.

A brief description of possible approaches to the represention and processing of KSAR queues follows. At the end of this section is a discussion of the current implementation of the KSAR queueing system in RTBB. Ideally, KSAR priorities should be dynamically determined by the threat the aircraft presents to the airspace, which is represented by origin of the coordinate system. For dynamic prioritization, the planner must contain rules for assessing the relative severity and immediacy of a threat. Futhermore, scheduling of the threatening tnodes must allow other goal nodes in the system to be serviced often enough so that future threats will not be ignored. Evidently, designing a planner and scheduler for such dynamic prioritization is a complex task and is not addressed in RTBB. A simpler approach to KSAR prioritization was chosen. The virtue of this approach is that the main BB process forks off KS computations, while the main process attends to other chores. To accomplish KSAR prioritization in RTBB, make a separate KSAR queue for each KS and then visit each queue, implemented with a FIFO access discipline, in a cyclic fashion, as shown in Figure 4a. The main consequence of this prioritization is that every goal node gets equal priority through its KSAR. In other words, the priority accorded a goal node does not depend on its abstraction level, as is the case with some other systems. This statement may appear excessively simplistic, but we felt that not enough knowledge was available about the RTP at this time for a more sophisticated approach.

The rule-based planner for mapping goal nodes into KSARs is a forward-chaining system and is based, in part, on the forward-chaining system in Reference 19. An example of a rule from the planner follows:

```
;;  Rule 5 creates a KSAR for invoking the MERGE-SEGMENTS KS if
;;   appropriate conditions are satisfied by the goal node.

    (setq  rule5
    '(rule   merge-segments
      (if
      (and
      (equal (send gnode :type) 'extend-segments) ; is it a purge node
      (null (send gnode :ksarptr)) ; no extend segment ksar active
      (setq  rvar1  (find-oldest-segment))
      (setq  rvar3  (find-most-recently-started-segment-with-length-gt-y  1))
      (setq rvar2 (abs (diff (car (send rvar1 :time))
         (car (last (send rvar3 :time))))))
      (and (> rvar2 3) (<= rvar2 10)) ; is age of proper range
      ))
      ;; --- rule attempts to patch fades in signal ---
      (then
      (progn   ; this creates ksar and sets ksarptr to that ksar
      (send  gnode  :set-ksarptr  (create-segment-merge-ksar  gnode))
      (format t "~% 5555555  CLOCK ~a  55555555555555555555 " clock)
      (format t "~%$$$$$ rule 5--- MERGE-SEGMENTS --- fired $$$$$")
    ))))
```

This rule states that

> IF—the goal node is of type extend-segments, and no KSARs are fired from this rule, and the difference between the end time of a segment and the start time of another segment is between three and 10 time units,
>
> THEN—create a KSAR to merge the two segments.

Note that this rule is disabled by the send gnode statement in the consequent of the rule by assigning the *ksarptr* to point to the generated KSAR, because the *ksarptr* must be nil to fire the rule. The format statements are merely to print out a history file on a BB run. The first format statement will print out a line such as "555555 CLOCK 7 5555555555", indicating that rule 5 was fired at clock time 7. The second format statement would similarly print out "$$$$$ rule 5 -- MERGE-SEGMENTS -- fired $$$$$$" on a new line.

The above rule creates a KSAR by a call to create-merge-segment-ksar function, which simply makes an instance of the KSAR flavor and then pushes this instance into the KSAR queue. This function is fairly easy and is shown as follows:

```
;;
;;  This function creates the merge-segments ksar
;;
(defun  create-merge-segment-ksar  (gnode)
  (sendksarpush
  (make-instance  'ksar
      :priority  1
      :ksar-id  'merge
      :ks nil
      :boot  '(merge-segments)
      :cycle  clock
      :context  gnode
      )
  ksarq)
  )
```

The following is an example of a KSAR created by a call to the above function.

```
;; An example of KSAR that seeks to invoke which MERGE-SEGMENTS KS

<ksar 1074948> is an instance of flavor ksar with instance variables:
   priority:        1
   ksar-id:         extension
   ks:              nil
   cycle:           40
   trigger:         nil
   context:         <bbevent 1071572>
   preconditions:   nil
   boot:            (merge-segments)
   nodeptr:         nil
   channel:         nil
   messenger:       nil
   command:         nil
   arglyst:         nil
   anslyst:         nil
   preboot:         nil
   prelyst:         nil
```

This KSAR is constructed by making an instance of the following flavor with the mixin ks-protocol, whose purpose should become clear when distributed KSARs are discussed.

```
(defflavor ksar      (
    priority  ;;       statis priority now
    ksar-id ;;         used at present
    ks    ;;           ks to be fired
    cycle  ;;          cycle created trigger
    context  ;;        arguments to the function boot
    preconditions ;;   undefined for now
    boot   ;;          the function call for the ks
    nodeptr  ;;        can point to any node
    channel  ;;        nil no transmission, -1 ready-to-read,
                       ;; 1 ready-to-write
    messenger  ;;      the i/o handler for this ksar
    )
(ks-protocol)
:gettable-instance-variables
:settable-instance-variables
:inittable-inst. nce-variables)
```

In the above KSAR, the variable *priority* needs some explanation. As stated earlier, a separate KSAR queue created for each KS was our goal in this research project, but this goal has not been fully achieved. At this time in RTBB, we have separate KSAR queues only for the distributed KSARs, those corresponding to the GETBEAM and GETASSIGNMENT KSs. The queue for the GETBEAM KS is called *beam-queue*, and the queue for the GETASSIGNMENT KS is called *assign-queue*. All the atomic KSARs are enqueued separately; this queue is called the *atomic-queue*. While the *beam-queue* and the *assign-queue* are FIFO, as they should be, imposing the same queueing discipline on the *atomic-queue* would be unreasonable. The instantiation of the variable *priority* reflects the priority that should be accorded to the KSAR shown in the *atomic-queue*.

The variable *ksar-id* is instantiated to a symbol that reflects the general activity of the KS invoked by the KSAR—in this case the activity is 'merge. The variable KS is usually instantiated to the previous activity that results in a data node. Addition of a data node to the data panel gives rise to the goal node, which leads to the present KSAR. Note that both these variables are not important to the processing of KSARs and have not been used in a consistent manner.

The variable *cycle* is instantiated to the clock time at which the KSAR was created. The variable *trigger* is not important to KSAR processing and should be ignored. The variable *context* is instantiated to the pertinent aspects of the context at the time of the KSAR creation. The instantiation for this variable can be as simple as

just the internal identity of the bbevent that caused the creation of the KSAR; or, in other cases, can include the latest time associated with an snode, the number of hits of which the snode is composed, etc.

The variable *preconditions* is not used at this time. Perhaps this variable could be used at a future date to ascertain if the conditions that give rise to the KSAR are still valid at the time the KS is fired. For such usage, *preconditions* are set to the minimum conditions to fire the KS to satisfy the goal node that gives rise to the KSAR.

The variable *boot* is important and is instantiated to the name of the KS that the KSAR must invoke. The KS is invoked by making a function call composed of the name of the KS, followed by appropriate arguments. The variable *nodeptr* is set to the internal identity of the goal node that gave rise to the KSAR. Other variables in this KSAR will be explained later in this section.

When a KSAR of the type mentioned above is selected and its corresponding KS executed,. then control resides completely in the KS during the KS processing. In other words, the main BB process waits for the KS to finish before focussing on any other activity. We call these KSARs atomic. In RTBB, atomic KSARs were used for most of the KSs. One advantage of an atomic KSAR is that it allows the KS to wrest control from the main BB process and implicitly freeze the context. In other words, because the information on the BB cannot alter during execution of the KS, you do not have to worry about the inapplicability of what might be returned by the KS. Clearly, if the information on the BB was allowed to change during execution of the KS, it is entirely possible that what is returned by the KS may not be relevant to the new state of the BB.

One major disadvantage of an atomic KSAR is that it does not permit exploitation of parallelism usually associated with BB problem solving. As mentioned in Section 1, one main attraction of using the BB paradigm is that the KSs, if representing independent modules of domain knowledge, should lend themselves to parallel invocation. Although parallel executions of KSs are highly desirable from the standpoint of enhancing performance, you should beware. Parallel execution also demands that attention be paid to the elimination of interference between the KSs, because one KS should not destroy the

conditions that must exist on the BB for the results returned by another KS to be relevant. Researchers have proposed methods to deal with these difficulties; the methods consist of either locking regions of the BB database or tagging different nodes with the identities of the KSs that need them [Reference 25]. Another opinion is that you should not bother with the overhead associated with region locking or data tagging, and should simply let the BB resolve any inconsistencies that might arise because of interference between the KSs [Reference 25].

In addition to atomic KSARs in RTBB, we have another type of KSAR that permits parallel invocation of two of the KSs; the latter type are called distributed KSARs. KSs that can be invoked via distributed KSARs are GET-BEAM and GETASSIGNMENT. An instance of a distributed KSAR is made from the same flavor used for an atomic KSAR. A most important characteristic of a distributed KSAR is that the BB interaction with the KS is allowed to take place on a polling basis.

The KS corresponding to a distributed KSAR is executed in three stages. The first stage sends a command to the KS with all the information needed to execute the KS. The format is just a list that represents a function call with all the information as arguments. The KS then just 'eval's the list. The second stage occurs when the system does a non-blocking read of the KS port to see if the KS is finished. A non-blocking read checks the port to see if data are available before actually reading the data. If we had used a regular read, for example via read or tyipeek functions, and no data were available at the port, the used function would wait indefinitely for the data to appear or do something unpredictable; but that is not what we wanted. We wanted to be able to poll the KS every few clock cycles, check whether or not the KS had returned the results, then read the results, if available. In the absence of results, we wanted the system to move on to other tasks, and return to the KS at a later time. Hence, the reason for non-blocking read. The non-blocking read function stores the KS results in the KSAR. The third stage occurs when the BB modifies the answer returned from the KS accordingly. Between stages, the BB actively works on other parts of the problem. The result is a speedup because of the parallel processing carried out by the system.

An example of a distributed KSAR that seeks to invoke the GETASSIGNMENT KS follows.

```
<ksar 1074284> is an instance of flavor ksar with instance variables:
      priority:          1
      ksar-id:           segment
      ks:                hit
      cycle:             40
      trigger:           change
      context:           ((time nil) (number <bbevent 1075036>) &)
      preconditions:     empty
      boot:              (post-assign-hits)
      nodeptr:           <bbevent 1075036>
      stage:             1
      messenger:         <messenger 1072204>
      command:           getassignment
        arglyst:         ('((8 93.54559999999999 6.019744 0.0)
                         (8 5.52 93.54559999999999 0.0)
                         (8 93.54559999999999 5.52 0.0))
                         '((9 92.67895 6.1425 0.0)
                         (9 6.1425 92.67895 0.0)
                         (9 92.67895 6.642136 0.0)))
   anslyst:              nil
   preboot:              (pre-assign-hits)
   prelyst:              ((<snode 1073184> <snode 1073144>
                         <snode 1072948>) & & & 9)
```

This KSAR is created by making an instance of the KSAR flavor shown earlier. The flavor ks-protocol, which is a part of the KSAR flavor definition, is presented below.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This is a mixing flavor called ks-protocol
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defflavor  ks-protocol
   (
   command     ;     the input function
   arglyst     ;     the argument list
   anslyst     ;     answer list
   preboot     ;     command to start up function after read
   prelyst     ;     argument list for reboot after read
   )

   :gettable-instance-variables
   :settable-instance-variables
   :inittable-instance-variables
   )
```

As will be evident from the following definitions of the variables, the mixin flavor is only useful for a distributed KSAR. Because all KSAR instances use the mixin, you might wonder why you should use the mixing ks-protocol at all; after all, the variables in the mixin could have been incorporated in the definition of the KSAR flavor. Note that even when a mixin is always used for defining objects, the mixin's separate definition allows the definitions of objects to be expanded incrementally as the software develops. Also, take advantage of the fact that mixin associated methods will be invoked in a certain order, depending on the order of appearance of mixins, etc.

The nature of the variables from *priority* through *nodeptr* in connection with atomic KSARs has already been explained. We will now define the other variables. The variable *stage* is instantiated to either 2, 1, -1, or 0. When the instantiation is 1, the KSAR is in the first stage, meaning that the KSAR is ready to send a command to the KS that would initiate execution of the KS. The command is taken off the variable *command* and the arguments from *arglyst*. After the *command* is transmitted to the KS, the instantiation of *stage* is set to -1, which is a signal to the BB process to start polling the KS port for new results using non-blocking read. The results are read off the KS port, deposited in the KSAR at *anslyst*, and the instantiation of *stage* is changed to 0. The instantiation of 0 for *stage* causes the function at *boot*, in this case post-assign-hits, to take the results out of the KSAR and deposit them at the appropriate place in the BB database, at which time the KSAR ceases to exist.* Obviously, *stage* is used to sequence initiation, execution, and results-reporting phases of KS operation in the correct order. In this KSAR example, the variable *arglyst* already has an instantiation, so KSAR processing can begin in stage 1. In some cases *arglyst* instantiation can be generated easily at the time the KSAR is formed by the planner—when a KSAR is formed to invoke GETBEAM because the *arglyst* here is nil. In other cases, some computational effort may have to be expended to construct the arguments. In the latter cases, *arglyst* is synthesized by adding yet another stage to the three stages already mentioned.

---

* The use of boot in a distributed KSAR is different from that in an atomic KSAR. In the latter, boot holds the function name to invoke the KS, a job now carried out by the instantiation of command. This inconsistency in the use of boot and some other variables is due to the manner in which RTBB has evolved.

This additional stage is specified by instantiating *stage* to 2. When the scheduler sees this instantiation, a function call is put out that constructs the arguments; the function call is held in the variable *preboot*. In the above example, the *arglyst* instantiation was generated by a call to the function (preassign-hits) when *stage* was set to 2. The *preboot* function, in this case pre-assign-hits, synthesizes arguments for the function call to the KS, and also puts together, for diagnostic purposes, a list of all the BB database items used for the arguments. The database items used are stored in the variable *prelyst*.

A note of explanation is in order for the exact nature of arguments under *arglyst* in the above example. The function pre-assign-hits examines all the snodes in the BB database and lists the most recent hit from each snode. This list of hits is the first of two arguments under the variable *arglyst*; the time stamp that corresponds to this argument is 8. The second argument under *arglyst*, which corresponds to time stamp 9, is the list of hit nodes that must be assigned to the segments or allowed to form new segments. The GETASSIGNMENT KS then tries to assign each new hit to a segment, which is based on the spatial and temporal closeness of the hit to the most recent entry.

The actual activation of a KS, for both the atomic and distributed KSARs, is carried out by sending a write command to a flavor that acts as an I/O handler for the BB. The write command is synthesized by the following method, which is defined for the KS-protocol flavor.*

```
;; This  method writes to the input port of the KS, which is the
;; same as one of the output ports of the BB process.

(defmethod (ks-protocol :write-ks) ()
(format t "COMMAND sent to ks ~a~%" (cons command arglyst))
(format (send  ; get output port name from messenger flavor
   (send self :messenger)  ; get messenger name from variable
   :write-port) "~a~%"
   (cons command arglyst)) ; form function call
(send self :set-stage -1)  ; change state of ksar to read
)
```

* Note that this method is neither an after-method nor a before-method. The method shown here is a primary method that is invoked by sending the ':write-ks' message to the ks-protocol flavor.

Essentially, a complex format statement that finds the correct input port to the KS (which is the same as an output port of the BB process), constructs the command sequence from the variables *command* and *arglyst* in the KSAR, and sends the command to the port. Before exiting, the method also changes the state of the KSAR *stage* to reflect that the command was sent to start KS execution, and that the KSAR is now ready for non-blocking read.

We have not yet explained the purpose of the variable *messenger* shown above. To understand the function of the *messenger* variable in the distributed KSAR example, associate an I/O handler with each KS. The handler should contain information such as the identity of the input and output ports associated with the KS. I/O handlers are created by making instances of the messenger flavor shown below.

```
;;  This is the flavor messenger

    (defflavor messenger     ;; these should be named after ks's
    (
        write-port  ;     the output port to the process
        write-fd    ;     the output port file descriptor
        read-port   ;     the input port to the process
        read-fd     ;     the input port file descriptor
        pid         ;     the process identity
        )
        ()
        :gettable-instance-variables
        :settable-instance-variables
        :inittable-instance-variables
    )
```

The variables *write-port* and *read-port* are instantiated to internally-generated symbolic names that designate the two ports; the symbolic names are returned by a '*process' call like

```
(*process 'path t t)
```

This call will return

```
(#<port from-process> #<port to-process> 13067)
```

where the symbolic name #<port from-process> is the output port of the unix process whose processor id is 13067, the unix process in this case being 'path'. Similarly, #<port to-process> is the symbolic name

of the input port of the process that calls up the UNIX process representing the KS. For the benefit of readers not familiar with the LISP-UNIX interface, a call like (*process 'path) would actually run the UNIX process 'path'. The variables *write-fd* and *read-fd* are instantiated to the file descriptors for the two ports; these file variables were convenient for diagnostics but are not used for anything at this time. The variable *pid* is instantiated to the process *id*; this variable also is not used at this time. The variable *messenger* in the distributed KSAR is instantiated to the identity of that instance of the messenger flavor associated with the KS that the KSAR seeks to invoke.

A discussion of how the KSARs are queued in the current implementation of RTBB follows. As previously mentioned, to maximize the potential for parallel implementations of the KSs, the system should construct separate KSAR queues for each KS. However, the current implementation has separate queues only for the GETBEAM and GETASSIGNMENT KSs, called *beam-queue* and *assign-queue*, respectively; all other KSARs are queued into the *atomic-queue* (Figure 4b). Each KSAR queue is an instance of the following event flavor:

```
(defflavor event (
        number
        (mask '(1 1 ))
        (atomic-queue  '())
        (beam-queue  '())
           (assign-queue  '())
        )
    ()
    :gettable-instance-variables
    :settable-instance-variables
    :inittable-instance-variables)
```

Note that the *mask* represents the status of the KSARs currently at the head of the queues. The *mask* instantiation list has a status value for each of the distributed-KSAR queues, and the interpretation given to each value in the list is the same as for instantiations of the variable *stage* in a distributed KSAR. In the defflavor, the initial mask values have been set to 1 for head KSAR in both the *beam-queue* and the *assign-queue*, meaning any KSARs found at the head of the respective queues are in stage 1. *Stage* value of 1 corresponds to *write* stage in which commands are written to the KSs.

A single instance of this flavor is made, and the resulting object is called *ksarq*. The variable *atomic-queue* of this object, initially a null list, is instantiated to the list of all the atomic KSARs; the variable *beam-queue* is instantiated to the list of all the distributed KSARs that seek to invoke the GETBEAM KS; and, finally, the variable *assign-queue* is instantiated to the list of all the distributed KSARs that seek to invoke the GETASSIGNMENT KS.

The RTBB scheduler cycles through the three queues, looks at the head KSAR in each queue, and services the KSAR in a manner that depends on whether the KSAR is in *atomic-queue* or one of the other queues. For the *atomic-queue* the KS is threaded into the BB process before visiting the other queues. For the *beam-queue* and *assign-queue*, the KS activation is executed in stages described earlier so that the BB does not wait for the KS to finish executing.

The following is an example of *ksarq* during execution.

```
<event 1071376> is an instance of flavor event with instance variables:
   number:        3
   mask:          (1 nil 2)
   atomic-queue:  (<ksar 1074140> <ksar 1074212>)
   beam-queue:    nil
   assign-queue:  (<ksar 1074284>)
```

The *mask* represents the status of the *atomic-queue* as 1, which means nothing for this queue; nil for the *beam-queue*, which is empty; and 2 for the *assign-queue*, which means the KSAR is ready for the *preboot* function to be run. The variable *number* is instantiated to the total number of KSARs held in the queueing system and updated at every change by a defmethod.

The clock is used in the system in the following way. Each cycle of the scheduler goes through all three queues. Each cycle of the scheduler is followed by an invocation of the planner, which maps all the previously unattended goals into KSARs or subgoals. One cycle of the scheduler followed by one invocation of the planner constitutes one control cycle, and one control cycle constitutes one clock unit. When the BB process is first started, the main control loop deposits a goal at the hit level; this goal, which generates new hits, is placed at the hit level every fourth clock unit. The scheduler now looks at all the queues, first examining the *atomic-queue*, which is

empty. The scheduler then examines the *beam-queue*, where a KSAR generated by the planner from the hit-level goal is found. The scheduler services this KSAR according to the *stage* status value stored in the *mask* variable of *ksarq*. Finally, the scheduler looks at the *assign-queue*, which is also empty. The process then repeats, as depicted in Figure 4b.

The main control loop, which alternately runs the planner and the scheduler, is shown below.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  this is the main loop for driving the BB
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    (defun cloop ()
    (do ()     ;put into infinite loop
    (())
    (go-for-it )    ; allows you to choose the number of control cycles
    (clock-update) ; updates the clock variable and place a goal at the
        ; hit level every fourth clock unit. It also places
        ; a purge-segments goal at the hit level every fourth
        ; cycle.
    (plan-goals)    ; maps the goals into ksars, it calls the rule-based
        ; planner
    (scheduler)       ; runs the scheduler which cycles through the three
        ; KSAR queues in ksarq.
    ))
```

The comments explain the nature of each function in the main control loop.

8
CONCLUSIONS

The purpose of this report is to convey a sense of how LISP object-oriented programming can be used to construct a BB. In practically all the literature we went through, details on how to program a BB were sorely missing. We hope this report rectifies that deficiency, at least to some extent.

Obviously, our BB was meant as a learning and training exercise. Therefore, our efforts should not be judged on whether we

succeeded in designing a usuable system for controlling a radar system. Our efforts should instead be judged on whether we succeeded in reducing the problem to manageable proportions, and in explaining the important details of our implementation.

Although the RTBB system currently works, many aspects of the system could be refined. For example, one of our future goals is to implement a separate queue for each KS, thus making it possible to use the RTBB in a parallel or multiprocessor mode. Another goal is to have all the KSARs be the distributed type, which would make it necessary to somehow split those KSs currently processed via atomic KSARs into pre, write, read, and post phases. The RTBB rule-based planner is rudimentary at this point. A more knowledgeable planner could be created to better focus the control.

As mentioned in the previous section, a clock unit in RTBB consists of the scheduler taking one pass through all the queues and one invocation of the planner. This definition of a clock unit makes the programming easy, although somewhat artificial. If the BB was run by a real clock, we would have to design functions to buffer the radar returns; the BB would then take the hits out of the buffer when able to address that task. Real-time implementation of RTBB is a future goal.

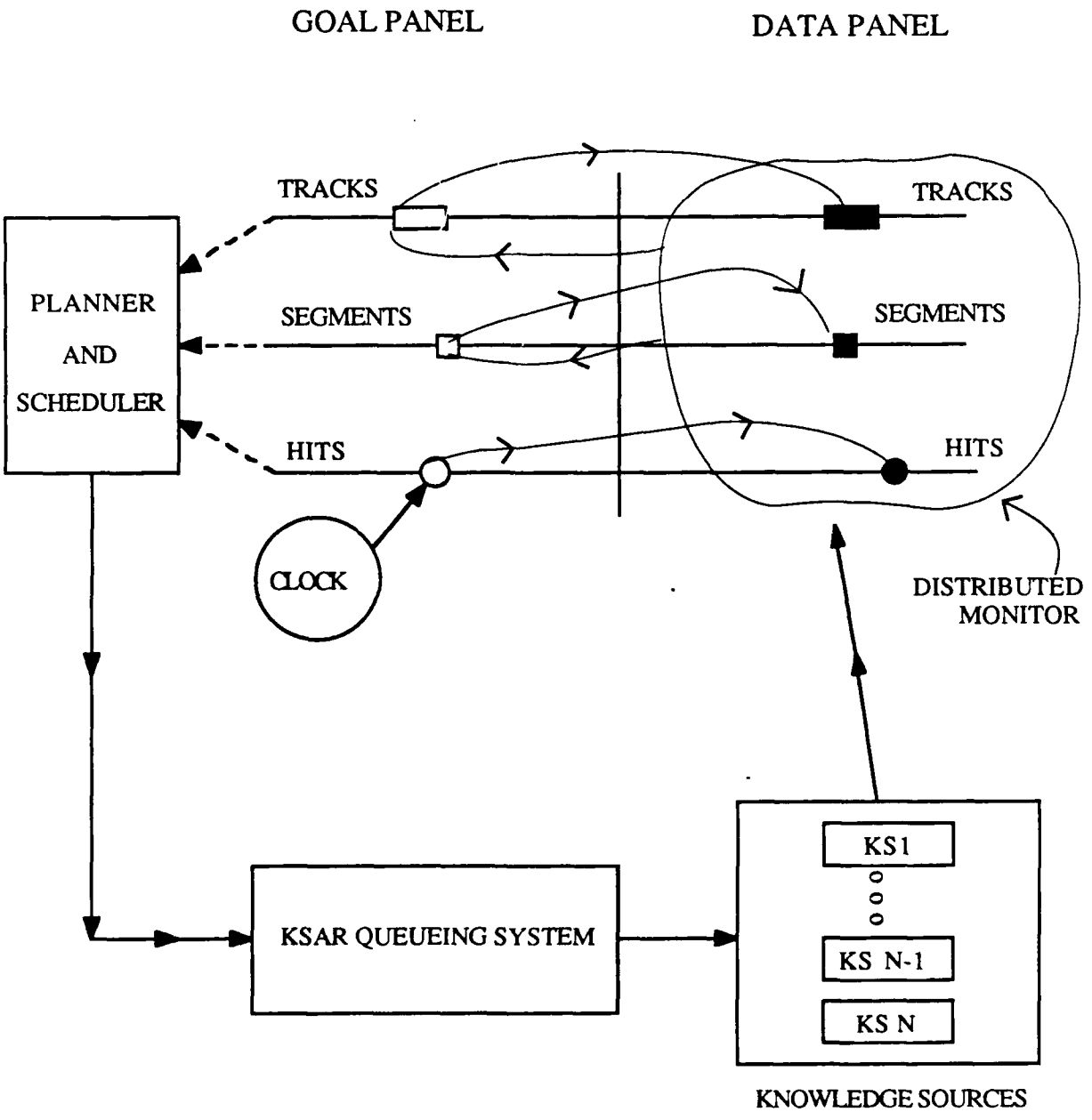
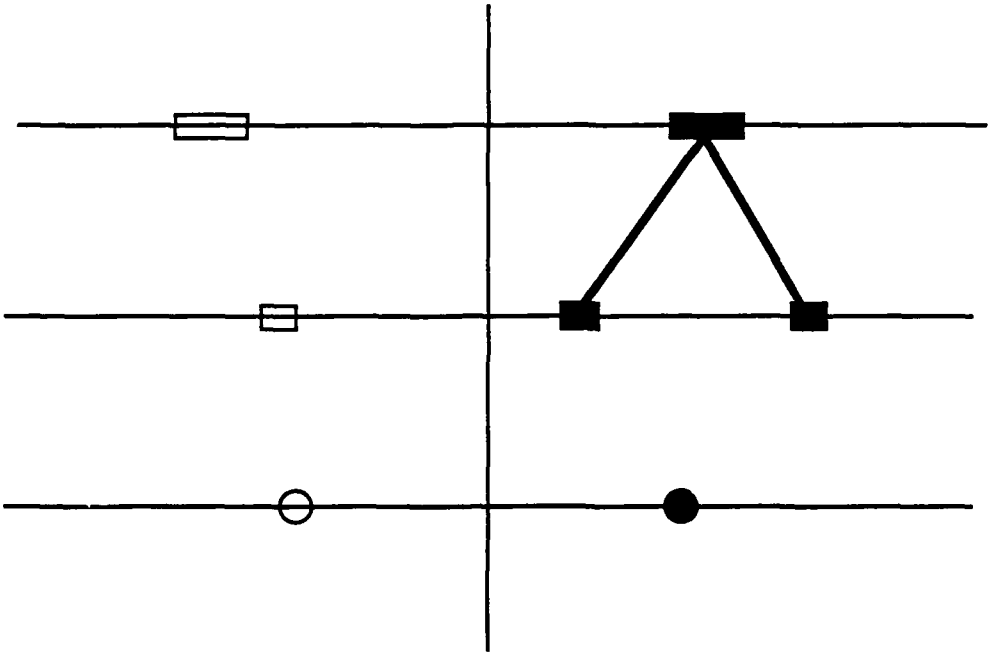


FIGURE 1. Radar Tracking Blackboard (BB).

GOAL PANEL DATA PANEL



Goal Nodes

○. Hits—flavors of type bbevent that express the desire to generate a data hit node.

▭ . Segments—flavors of type bbevent that express the desire to match the hit data to segments.

▭ . Tracks—flavors of type bbevent that express the desire to establish the segment node as part of a track.

Data Nodes

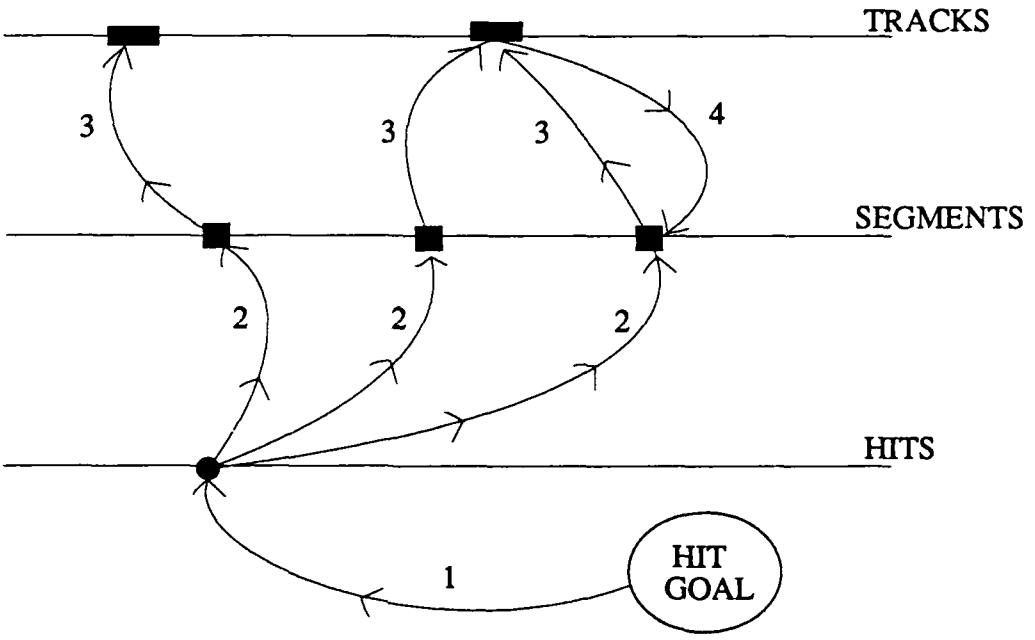
●. Hits—flavors of type bnode that contain the returns information.

■. Segments—flavors of type snode that contain the linear fits.

▬. Tracks—flavors of type tnode that contain the groups of close segments.

(a) Blackboard goal and data node descriptions.

FIGURE 2. Blackboard Nodes and Knowledge Sources.



1. HIT GENERATION - simulates aircraft trajectories.

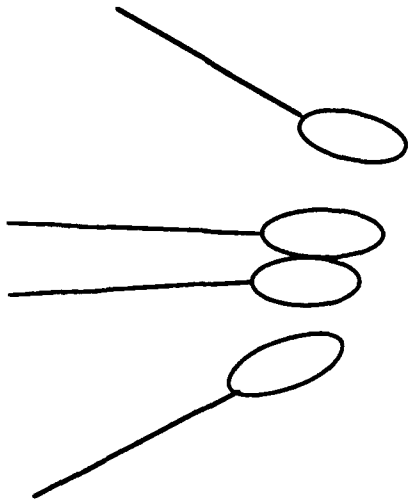
2. ASSIGNMENT - associates radar returns to best segments.

3. TRACK FORMATION - associates individual segments with tracks.

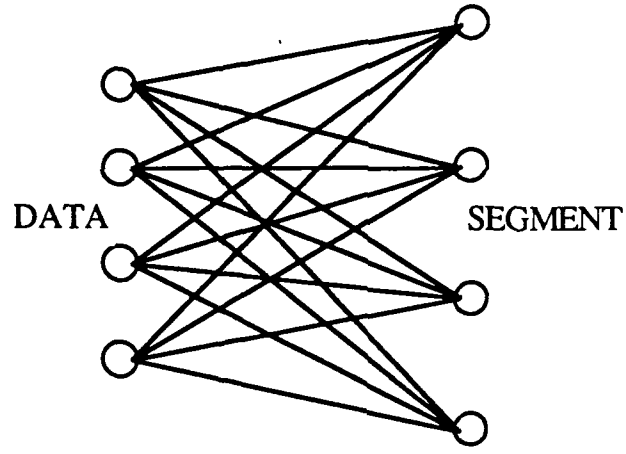
4. SPLINE - checks if segments are associated with the proper tracks.

(b) General description of knowledge sources.

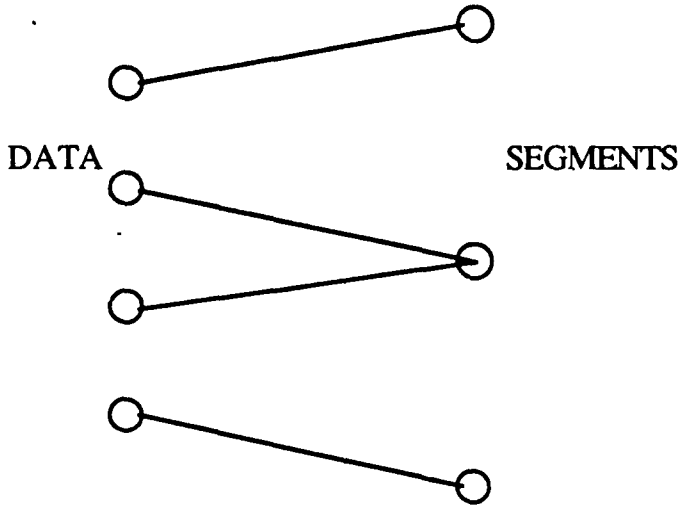
FIGURE 2. (Contd.)



(a) Actual tracks

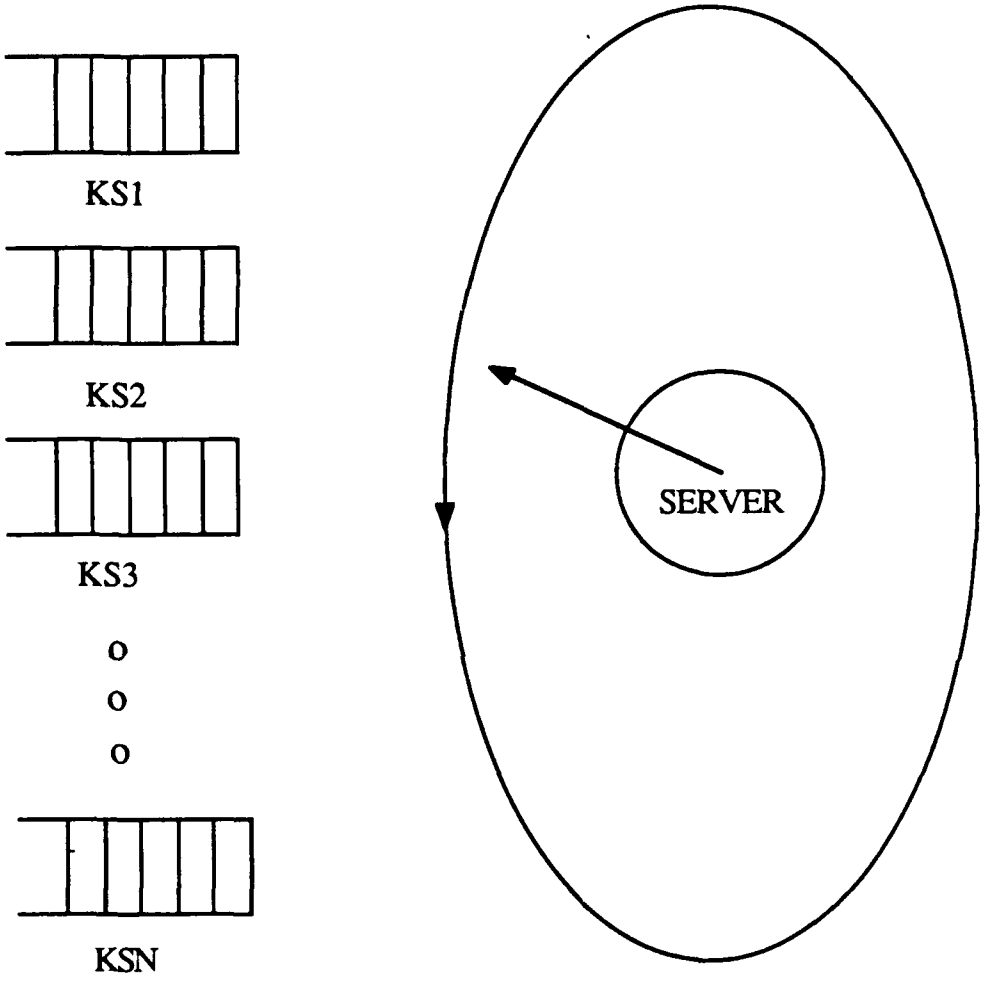


(b) Possible associations.



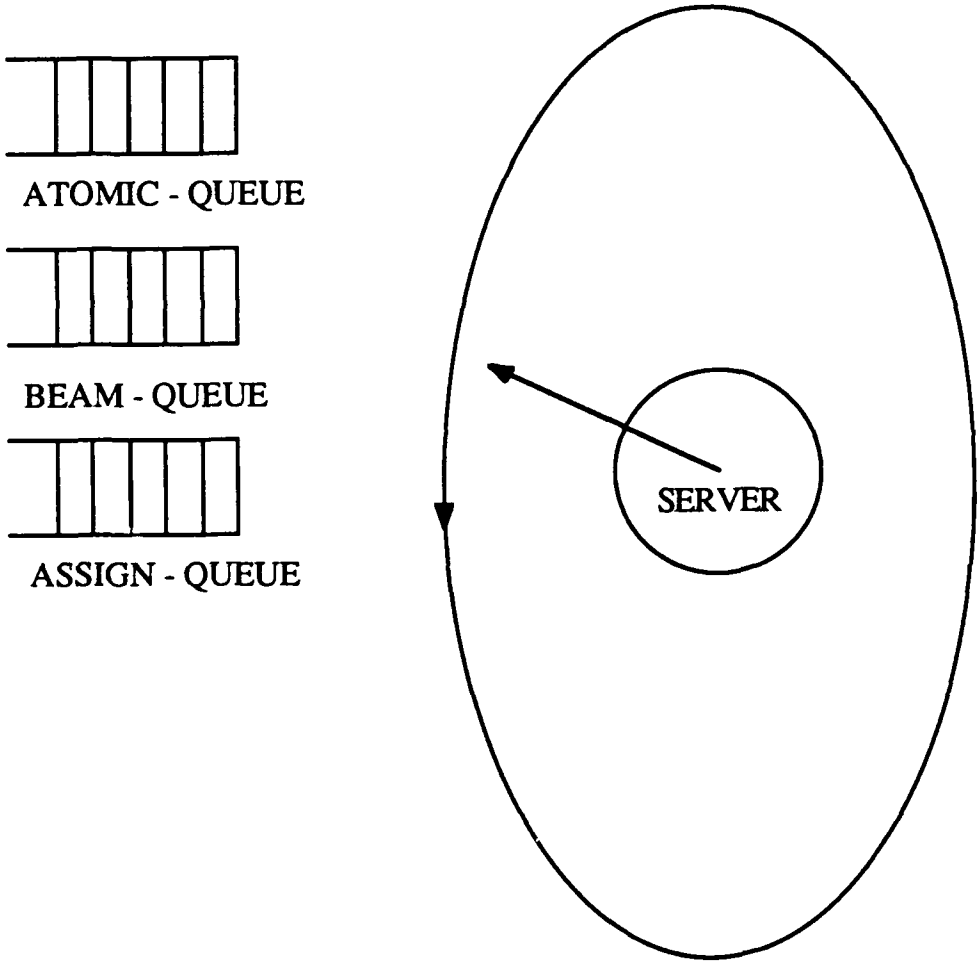
(c) Actual associations.

FIGURE 3. The Assignment Problem and Its Graphical Interpretation.



(a) Ideal KSAR queueing system.

FIGURE 4. KSAR Queueing System. The server is the BB process. The queueing discipline is cyclic, with at most one task or subtask executed per server visit.



(b) Actual KSAR queueing system.

FIGURE 4 (Contd.)

REFERENCES

1. Yaakov Bar-Shalom and Thomas E. Fortmann. *Tracking and Data Association.* New York, Academic Press, 1987. Pp. 1-353.

2. V. Lesser and D. Corkill. "Funtionally Accurate, Cooperative, Distributed Systems," *IEEE Transactions on Systems, Man, & Cybernetics*, Vol. SMC-11, No. 1. (January 1981), pp. 81-96.

3. H. Penny Nii. "Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures," *The AI Magazine*, Summer 1986, pp. 38-53.

4. H. Penny Nii. "Blackboard Systems Blackboard Application Systems, Blackboard Systems from a Knowledge Engineering Perspective," *The AI Magazine*, August 1986, pp. 82-106.

5. *Blackboard Systems*, ed. by Robert Englemore and Tony Morgen. New York, Addison-Wesley, 1988. Pp. 1-602.

6. H. Penny Nii and others. "Signal-to-Symbol Transformation: HASP/SIAP Case Study," *The AI Magazine*, Spring 1982, pp. 23-35.

7. K. M. Andress and A. C. Kak. *AI Magazine*, Summer 1988, pp. 75-94.

8. K. Andress and A. C. Kak. "The PSEIKI Report - Version 2" in *Technical Report TR-EE 88-9,* Purdue University, W. Lafayette, Ind., 1988.

9. Daniel D. Corkill. *A Framework for Organizational Self-Design in Distributed Problem Solving Networks.* Ph.D. Thesis, U of Mass, February 1983, pp. 1-372.

10. Barbara Hayes-Roth. "A Blackboard Architecture for Control," *Artificial Intelligence*, Vol 26, (1985), pp. 251-321.

11. Victor Lesser and Edmund Durfee. "Incremental Planning in a Blackboard-Based Problem Solver," in proceedings of the National Conference on Artificial Intelligence, AAAI Conference, Philadelphia, Pa., August 1986. Pp. 58-64.

12. "Visions: a Computer System for Interpreting Science," in *Computer Vision Systems*, ed. by A. R. Hanson and E. M. Riseman. New York, Academic Press, 1978. Pp. 303-33.

13. M. A. Williams. "Distributed, Cooperating Expert Systems for Signal Understanding," in proceedings of Seminar on AI Applications to Battlefield, 1985. Sections 3.4-1 to 3.4-6.

14. R. Worden. "Blackboard Systems," in *Computer Assisted Decision Making*, ed. by G. Mitra. New York, North Holland, 1986. Pp. 95-106.

15. Naval Weapons Center. *Distributed Inference Structures Bids and Proposals Report*, by P. R. Kersten and J. L. Hodge. China Lake, Calif., NWC, September 1986. 16 pp. (NWC TM 5857, publication UNCLASSIFIED.)

16. Daniel D. Corkill and others. *GBB: A Generic Blackboard Development Systems*, in proceedings of the Fifth National Conference on Artificial Intelligence, AAAI Conference, 1986, Philadelphia, Pa., 1986. Pp. 1008-14.

17. I. D. Craig. "The Ariadne-1 Blackboard System," *The Computer Journal*, Vol. 29, No. 3 (1986), pp. 235-40.

18. *Franz LISP Reference Manual*, ECN No. 750. Purdue University, W. Lafayette, Ind., March 1987.

19. P. H. Winston and B. K. P. Horn. *LISP*, 2nd ed. New York, Addison-Wesley, 1984. Pp. 1-434.

20. Robert B. Cooper. *Introduction to Queueing Theory*. 2nd ed., New York, North Holland, 1981. Pp. 1-347.

21. I. Faux and M. Pratt. *Computational Geometry for Design and Manufacture*. Ellis Horwood Limited, 1979.

22. Frederick S. Hillier and Gerald J. Lieberman, "Integer Programming," in *Introduction to Operations Research.* Oakland, Calif., Holden-Day, 1980.

23. Nils J. Nilsson. *Principles of Artificial Intelligence.* Palo Alto, Calif., Tioga Publishing Co., 1980. Pp. 1-476.

24. G. Forsythe, M. Malcolm, and C. Moler. *Computer Methods for Mathematical Computations..* New York, Prentice-Hall, Inc., 1977. Pp. 1-229.

25. V. Lesser and R. Fennell. "Parallelism in Aritificial Intelligence Problem Solving: A Case Study of Hearsay II," *IEEE Transactions on Computers*, Vol. C-26, No. 2, (February 77), pp. 98-143.

Appendix
EXAMPLES OF BLACKBOARD TRACKING

EXAMPLE 1

In this example, most of the flavors are expanded to illustrate the detail of each step. This expansion will be done only in this example.

This example illustrates the BB solution formation for a single trajectory. The data that drive the trajectory are based on Bezier's curve. The trapezoid that defines the trajectory is given by the four points indicated in Figure A-1. Note that the origin is one of the points, so the trajectory will go through the origin. The origin in these examples is a special point and represents not only the origin of the coordinate system, but the center of the air space around a hypothetical airport. The starting point of the single trajectory is (100,0,0).

The data bnode flavor for the initial and all other points is initiated periodically by placing a goal node on the hit level of the goal panel. The goal node generates a KSAR that fires the hit generation KS GETBEAM. As mentioned in Section 7 of this report, this KSAR is distributed, although no *preboot* function is necessary becuase the function call is so simple. The *preboot* may be considered a precondition type of function that freezes the context and extracts the variables needed by the KS. The *preconditions* variable is never used, a misnomer arising out of the development process. The KSAR looks as follows:

```
<ksar 1072368> is an instance of flavor ksar with instance variables:
   priority        2
   ksar-id:        newhit
   ks:             add
   cycle:          1
   trigger:        clock
   context:        none
   preconditions:  empty
   boot:           (getbeam)
   nodeptr:        nil
   channel:        1
   messenger:      <messenger 1072044>
   command:        fire
   arglyst:        nil
   anslyst:        nil
   preboot:        nil
   prelyst:        nil
```

The *boot* is the C-coded KS called GETBEAM and creates the command to fire the KS. The KSAR causes the formation of a data node to be placed on the hit level of the data panel. The data node looks as follows:

```
<bnode 1072668> is an instance of flavor bnode with instance variables:
   type:           hit
   time:           0
   coord:          ((100.0 0.0 0.0))
   number:         1
```

Note that the return count is given by *number* and occurs at *time* 0 at the coordinates *coord*. The node type is specified by *type*.

The placement of this hit node on the data panel causes placement of a goal node on the segment level of the goal panel. This goal node looks as follows:

<bbevent 1072808> is an instance of flavor bbevent with instance variables:

source:	<bnode 1072668>
action:	change
type:	hit
variable:	coord
time:	0
coord:	((100.0 0.0 0.0))
number:	1
threat:	nil
snode:	nil
pattern:	nil
duration:	one-shot
position:	nil
goalptr:	nil
conditions:	nil
ksarptr:	nil

This goal represents the desire to match this data to the nearest segments. The duration of the goal node is one-shot, which means the rule base gets only one pass to satisfy itself; otherwise the goal node is removed from the goal panel. The source is a pointer to the data node responsible for the creation of the goal node by the distributed monitor.

The rule base causes the segment goal node to generate a KSAR to match the hit data to the nearest segment, if there is one. Otherwise, KS creates a new segment. Again, the *ksar-id* generally describes the driving activity, i.e., segment formation. The KSAR for this KS is distributed with a separate *preboot* function that forms the arguments for the BB formation. The *boot* function is really the *postboot* function; use of the function changed while the BB system evolved, and the term *boot* became too embedded to easily change. The *command* variable carries the main KS call function and looks as follows:

```
<ksar 1072876> is an instance of flavor ksar with instance variables:
   priority:       1
   ksar-id:        segment
   ks:             hit
   cycle:          4
   trigger:        change
   context:        ((time nil)(number <bbevent 1072808>) &)
   preconditions:  empty
   boot:           (post-assign-hits)
   nodeptr:        <bbevent 1072808>
   channel:        2
   messenger:      <messenger 1072204>
   command:        getassignment
   arglyst:        nil
   anslyst:        nil
   preboot:        (pre-assign-hits)
   prelyst:        nil
```

The GETASSIGNMENT KS, which is fired by this KSAR, matches the segments to the data, and the post-assign-hits function places the nodes on the data panel at the segments level. This segment node looks as follows:

```
<snode 1072948> is an instance of flavor snode with instance variables:
   type:           segment
   time:           (0)
   coord:          ((100.0 0.0 0.0))
   number:         nil
   cpa:            nil
   linear:         nil
   tnode:          nil
   threat:         nil
```

Most of the variables are initially nil because the segment is not long enough. However, these variables will fill in later when the segment becomes part of a track. In fact, two time units later the snode looks as follows:

```
<snode 1072948> is an instance of flavor snode with instance variables;
   type:          segment
   time:          (1,0)
   coord:         ((99.24254999999999 0.7425 0.0) (100.0 0.0 0.0 ))
   number:        2
   cpa:           (49.00339911339883 49.99006688005953 0.0)
   linear:        ((99.24254999999999 0.7425 0.0 )
                  (-0.7574500000000057 0.7425 0.0 ))
   trode:         nil
   threat:        nil
```

At the formation of a data-segment node, a demon from the distributed monitor creates a track goal node that represents the desire to form a track from the segments. The goal node looks as follows:

```
<bbevent 1073816> is an instance of flavor bbevent with instance
variables:
   source:        <snode 1072948>
   action:        change
   type:          segment
   variable:      number
   time:          (1 0)
   coord:         ((99.24254999999999 0.7425 0.0)
                  (100.0 0.0 0.0))
   number:        2
   threat:        nil
   snode:         nil
   pattern:       nil
   duration:      one-shot
   position:      nil
   goalptr:       nil
   conditions:    nil
   ksarptr:       nil
```

Note that a data-segment node is the *source* of this node, and the *coord* is the two consecutive coordinates used to form the segment and the track. The *time* variable is the sequence of times that support formation of the track.

Again, the rule base creates the following KSAR from this track goal node. The purpose of the KSAR is to form tracks from the segments.

```
<ksar 1074640> is an instance of flavor ksar with instance variables:
   priority:       0
   ksar-id:        track
   ks:             segment
   cycle:          16
   trigger:        change
   context:        ((time nil)(number <snode 1072948>) &)
   preconditions:  empty
   boot:           (assign-tracks)
   nodeptr:        <snode 1072948>
   channel:        1
   messenger:      nil
   command:        nil
   arglyst:        nil
   analyst:        nil
   preboot:        nil
   prelyst:        nil
```

Unlike the previous KSAR on the assignment of hits, this KSAR is an atomic KSAR. The KS places a track node on the data panel at the track level. This node looks as follows:

```
<tnode 1074228> is an instance of flavor tnode with instance variables:
   type:           track
   time:           (1)
   last-coord:     (99.24254999999999 0.7425 0.0)
   last-velocity:  (-0.574500000000057 0.7425 0.0)
   threat:         nil
   snode:          (<snode 1072948>)
   cpa-bracket:    ((43.97948402473871 54.02731429295895)
                    (45.063561019205357 54.91482356806549))
   check:          nil
   checklyst:      nil
```

This data node *time* variable contains only the current time. The *last-coord* and *last-velocity* are the corresponding position and velocity. The variable snode contains a list of pointers to the segments that form the support for the tracks. The confidence region, which is called *cpa-bracket*, causes the *threat* variable to be marked if it includes the origin. For the node above, the track does not yet appear as a threat.

The above nodes are the initial formation of the solution track. The solution track structure is a tree; the base of the tree is the track node, and the branches are the segment nodes. This example has only one branch, so the solution tree is very simple. The track coordinate history contained in the tree expands as the track grows in length. As an example, consider a segment node at a later time:

```
<snode 1072984> is an instance of flavor snode with instance variables:
   type:          segment
   time:          (4 3 2 1 0)
   coord:         ((96.8832 2.88 0.0)
                  (97.68385000000001 2.1825 0.0)
                  (98.4704 1.47 0.0)
                  (99.24254999999999 0.7425 0.0)
                  (100.0 0.0 0.0))
   number:        5
   cpa:           (43.2289227832382 49.621934519913962 0.0)
   linear:        ((96.88322.88 0.0)
                  (-0.8006500000000045 0.6974999999999998 0.0))
   tnode:         <tnode 1074228>
   threat:        nil
```

Note that the *cpa* and the linear model that the segment included were calculated. The *threat* was evaluated and the track node, which this segment node supports, is contained in tnode. Information was sent to this node by demons associated with these nodes or by the KSs themselves.

After the segment information is extended to more than 13 points, the list is truncated with after-method. After about 20 time units, the snode looks as follows:

```
<snode 1072948> is an instance of flavor snode with instance variables:
   type:          segment
   time:          (20 19 18 17 16 15 14 13 12 11 10 9 8)
   coord:         ((82.40000000000001 12.0 0.0)
                  (83.38545000000001 11.5425 0.0) (84.3616 11.07 0.0)
                  (85.32814999999999 10.5825 0.0) (86.2848 10.08 0.0)
                  (87.23125 9.5625 0.0)
                  (88.16719999999999 9.029999999999999 0.0)
                  (89.09235 8.4825 0.0) (90.0064 7.92 0.0)
                  (90.90904999999999 7.3425 0.0) (91.8 6.75 0.0)
                  (92.67895 6.1425 0.0) (93.54559999999999 5.52 0.0))
   number:        13
   cpa:           (19.19401128389733 41.34369053489974 0.0)
   linear:        ((82.40000000000001 12.0 0.0)
                  (-0.9854500000000002 0.4574999999999996 0.0))
   tnode:         <tnode 1074228>
   threat:        nil
```

In the above snode, the maximum length of the *coord* and *time* variables is now only of length 13, as fixed by a global variable. The truncation length may be set to any fixed value, but this variable is not totally independent of the other parameters. For example, a

track may only be generated when the segment length exceeds another fixed parameter. Certainly, the truncation length must exceed this minimum length needed to initiate a track. Therefore, the truncation length must be chosen carefully, otherwise, the entire program could be comfounded.

The general sequence of KS calls is outlined in Figure A-2. The order of KS calls is numbered to push data nodes to higher levels of abstraction. The order is not exact because several data nodes must be advanced to form a track, but the general order required to push data through to support a track solution is outlined. The first KS is the hit generation KS (GETBEAM), the second KS is the GETASSIGMENT KS, and the third KS is the track formation KS (GETTRACK). Demons from the distributed monitor generate the goal nodes from the data nodes. The simple construction illustrated is essentially data driven with goal nodes isomorphically mapped to KS. This example illustrates the operation of a goal-driven BB emulating a data-driven BB.

EXAMPLE 2

In this example three separate craft are being observed. Three craft generate returns but only two track solutions are formed. This example illustrates the track-formation process and especially the grouping of segments into tracks.

Figure A-3 shows a plot of three trajectories. Two of these trajectories are very close and logically form a track. The other trajectory forms a separate track. The plots of Figure A-3 are mirrored in the data structures on the BB panels. Because a tree represents a track, one tree represents the two close trajectories, and the other tree represents the single trajectory. Each tree groups the time sequences to form a track, and the set of all tracks formed from the data is a forest.

Figure A-4 traces the formation of the solution trees on the BB. This formation is similar to that of a single track. The overall crisscrossing of the solution path on the BB panels from lower levels to higher levels of abstraction is because of the data-driven nature of the problem. The presence of three distinct trajectories in the data causes formation of three distinct nodes on the goal panel. Each goal represents the desire to use the data-segment node as support for a track node. Support means the segment node supports the hypothesis that the track node should contain that segment as part of the group that makes up the track.

The following two tracks show some of the data nodes on the BB after the tracks have been established:

```
<tnode 1074720> is an instance of flavor tnode with instance variables:
  type:           track
  time:           (5)
  last-coord:     (96.07797734375001 3.905422931640625 0.0)
  last-velocity:  (-0.8144500000000079 0.6824700000000004 0.0)
  threat:         nil
  snode:          (<snode 1072948> <snode 1073184>)
  cpa-bracket:    ((36.18550040607643 54.2522089149583)
                   (44.94766067922566 55.14530386696201))
  check:          nil
  checklyst:      nil

<tnode 1074676> is an instance of flavor tnode with instance variables:
  type:           track
  time:           (5)
  last-coord:     (3.6225 96.12575 0.0)
  last-velocity:  (0.6825000000000001 -0.8144500000000079 0.0)
  threat:         nil
  snode:          (<snode 1073144>)
  cpa-bracket:    ((44.80414791166273 54.91482356806549)
                   (35.91704278661875 54.02731420205895))
  check:          nil
  checklyst:      nil
```

Note that <tnode 1074720> is the second track in Figures A-3 and A-4 with two supporting segment nodes. Snode is the list of pointers to segment nodes. The pointers contained in snode are branches of the solution tree and the supports for the track hypothesis. The other track node <tnode 1074676> has only one pointer, which means it has only one branch and one supporting segment node. Neither track is presently a threat to the origin, although the track's plot indicates that this will not be true in the future.

In this case the snodes also contain parent pointers that establish which track they support. These nodes look as follows:

```
<snode 1073184> is an instance of flavor snode with instance variables:
   type:          segment
   time:          (5 4 3 2 1 0)
   coord:         ((96.06874999999999 4.062438 0.0)
                  (96.8832 3.379968 0.0)
                  (97.68385000000001 2.682486 0.
   number:        6
   cpa:           (41.62943218734221 49.67997281196416 0.0)
   linear:        ((96.06874999999999 4.062438 0.0)
                  (-0.8144500000000079 0.6824700000000004 0.0))
   tnode:         <tnode 1074720>
   threat:        nil

<snode 1073144> is an instance of flavor snode with instance variables:
   type:          segment
   time:          (5 4 3 2 1 0)
   coord:         ((3.5625 96.06874999999999 0.0) (2.88 96.8832 0.0)
                  (2.1825 97.68385000000001 0.0) (1.47 98.4704 0.0)
                  (0.7425 99.24254999999999 0.0) (0.0 100.0 0.0))
   number:        6
   cpa:           (49.38655323518081 41.38537980601705 0.0)
   linear:        ((3.5625 96.06874999999999 0.0)
                  (0.6825000000000001 -0.8144500000000079 0.0))
   tnode:         <tnode 1074676>
   threat:        nil

<snode 1072948> is an instance of flavor snode with instance variables:
   type:          segment
   time:          (5 4 3 2 1 0)
   coord:         ((96.06874999999999 3.5625 0.0)
                  (96.8832 2.88 0.0)(97.68385000000001 2.1825 0.0)
                  (98.4704 1.47 0.0) (99.24254999999999 0.7425 0.0)
                  (100.0 0.0 0.0))
   number:        6
   cpa:           (41.38537980601705 49.38655323518081 0.0)
   linear:        ((96.06874999999999 3.5625 0.0)
                  (-0.8144500000000079 0.6825000000000001 0.0))
   tnode:         <tnode 1074720>
   threat:        nil
```

At a much later time both tracks are classified as threats; at the time stamp of 41 the track nodes look as follows:

```
<tnode 1074720> is an instance of flavor tnode with instance variables:
   type:           track
   time:           (41)
   last-coord:     (60.0895833333333418.3031817537037  0.0)
   last-velocity:  (-1.111450000000005  0.1400399999999991  0.0)
   threat:         t
   snode:          (<snode 1072948> <snode 1073184>)
   cpa-bracket:    ((-2.437209695431297  54.2522089149583)
                    (24.69991086768157  55.14530386696201))
   check:          t
   checklyst:      nil

<tnode 1074676> is an instance of flavor tnode with instance variables:
   type:           track
   time:           (41)
   last-coord:     (18.7425  60.44255  0.0)
   last-velocity:  (0.1424999999999983  -1.111450000000005  0.0)
   threat:         t
   snode:          (<snode 1073144>)
   cpa-bracket:    ((24.69991086768157  54.91482356806549)
                    (-2.422621460220589  54.02731420205895))
   check:          nil
   checklyst:      nil
```

At the time stamp of 41, both tracks represent threats to the origin, and the threat variable is instantiated as true. The confidence region represented by the *cpa-bracket* has one coordinate that straddles the origin. This condition, although an arbitrary and probably not a sharp criterion, illustrates the detection via the rule-based system.

EXAMPLE 3

In this example, three separate craft are being observed. Initially, these three craft form one track. Subsequently, one craft breaks away from the established track. This example illustrates the detection of the breakaway and the subgoaling needed to establish two tracks.

Figure A-5 shows a plot of three trajectories. Initially, all these trajectories are very close and logically form a track. However, as the track evolves in time, one of the segments supporting the track formation departs from the track itself, meaning that if the track grouping were reformed, two tracks instead of one would be formed. The spline test is a back-chaining algorithm designed to detect whether the grouping of segments into a track is still logically valid.

One way to solve the problem of regrouping the tracks is to dissolve the track node, and keep the segment nodes on the data level after removing their parent pointers to a track. The track-formation algorithm would then pick up these uncommitted segments and regroup them into tracks. This solution is acceptable but not as desirable as maintaining the track history and forming a new track from a subset of the segments of the original track. This formation is implemented by subgoaling—an important technique that allows finer granularity in KSs and easier implementation of more complex goal interrelationships.

The nodes or solution tree should reflect the history of the trajectory. Indeed, Figure A-6 shows the parallel between the physical trajectories and data structures that represent these trajectories. First, a tree is formed on the BB that has only one root, i.e., one trajectory with three branches representing the three distinct craft. Once the track is established and determined to be a threat, the track grouping is checked via the spline KS. When the track grouping is not verified by the spline KS, subgoals for each

segment are created and placed on the goal segment level. Each goal represents the desire to determine if that segment is in the same equivalence class as the average track that represents the root of the track. If the segment does not satisfy the grouping criterion against the track, the segment is spun off as a segment with no parent pointers, which means the BB will establish this segment as a separate track. The following paragraphs show the state of the nodes in this sequence.

Initially, the track node formed from the three segments looks as follows:

```
<tnode 1074676> is an instance of flavor tnode with instance variables:
   type:           track
   time:           (1)
   last-coord:     (99.24254999999999 98.61066633333331 0.0)
   last-velocity:  (-0.7574500000000057 -1.722667000000001 0.0)
   threat:         nil
   snode:          (<snode 1072948> <snode 1073144)

<snode 1073184>)
   cpa-bracket:    ((32.08760233000798 62.90600256325121)
                   (-31.94628530341367 -7.667384301036718))
   check:          nil
   checklyst:      nil
```

Three snodes or branches support this track. The three segments supporting the trajectory are given below. The track-node pointers are really the parent pointers, or the edges of the graph, pointing to the root of the tree that represents the track.

```
<snode 1073184> is an instance of flavor snode with instance variables:
   type:          segment
   time:          (1 0)
   coord:         ((99.24254999999999  99.522324 0.0)
                  (100.0 101.0 0.0))
   number:        2
   cpa:           (38.19259757273452 -19.5773518900408 0.0)
   linear:        ((99.24254999999999  99.522324 0.0)
                  (-0.7574500000000057 -1.477676000000002 0.0))
   tnode:         <tnode 1074676>
   threat:        nil

<snode 1073144> is an instance of flavor snode with instance variables:
   type:          segment
   time:          (1 0)
   coord:         ((99.24254999999999  98.522325 0.0)
                  (100.0 100.0 0.0))
   number:        2
   cpa:           (38.59849373098595 -19.78542580508939 0.0)
   linear:        ((99.24254999999999  98.522325 0.0)
                  (-0.7574500000000057 -1.477675000000005 0.0))
   tnode:         <tnode 1074676>
   threat:        nil

<snode 1072948> is an instance of flavor snode with instance variables:
   type:          segment
   time:          (1 0)
   coord:         ((99.24254999999999  97.78735 0.0)
                  (100.0 100.0 0.0))
   number:        2
   cpa:           (58.86860840361245  -20.15231845764879 0.0)
   linear:        ((99.24254999999999  97.78735 0.0)
                  (-0.7574500000000057  -2.212649999999996 0.0))
   tnode:         <tnode 1074676>
   threat:        nil
```

This solution-tree structure is the initial state of the track before discovery that the trajectory is a threat and before departure of one of the craft from the formation.

At the time stamp of 11, the track is determined to be a threat to the origin, and the spline KS (GETSPLINE) now checks to see if composition of the track still makes sense. The following track node illustrates the track-node state just after threat determination.

```
<tnode 1074676> is an instance of flavor tnode with instance variables:
   type:           track
   time:           (11)
   last-coord:     (90.91376606802292 86.24495864263466 0.0)
   last-velocity:  (-0.8909500000000037 -1.085756000000003 0.0)
   threat:         t
   snode:          (<snode 1072948> <snode 1073144> <snode 1073184>)
   cp-bracket:     ((3.746220505494785 62.90600256325121)
                    (-32.05750115050004 0.08018509395760631))
   check:          nil
   checklyst:      nil
```

The spline test detects the breakaway of a track, then marks the track node *check* variable as failed. A failed spline test automatically disables further spline tests for that track until a track-verification KS can be run. The rule base will detect a failed track in the goal BB, then generate a subgoal for each segment that supports the track. Each goal expresses the desire to reevaluate the track-formation grouping criterion of each segment against the averaged track. The following are the subgoals generated by the rule base.

```
<bbevent 1074788> is an instance of flavor bbevent with instance
variables:
      source:         <tnode 1074676>
      action:         verify-track
      type:           track
      variable:       nil
      time:           (12)
      coord:          ((90.00946578449609 82.81725806347009 0.0)
                       (-0.9026499999999942 -1.052371999999991 0.0))
      number:         nil
      threat:         nil
      snode:          <snode 1073184>
      pattern:        nil
      duration:       one-shot
      position:       nil
      goalptr:        nil
      conditions:     nil
      ksarptr:        nil
```

<bbevent 1074720> is an instance of flavor bbevent with instance variables:

source:	<tnode 1074676>
action:	verify-track
type:	track
variable:	nil
time:	(12)
coord:	((90.00946578449609 82.81725806347009 0.0) (-0.9026499999999942 -1.052371999999991 0.0))
number:	nil
threat:	nil
snode:	<snode 1073144>
pattern:	nil
duration:	one-shot
position:	nil
goalptr:	nil
conditions:	nil
ksarptr:	nil

<bbevent 1075076> is an instance of flavor bbevent with instance variables:

source:	<tnode 1074676>
action:	verify-track
type:	track
variable:	nil
time:	(12)
coord:	((90.00946578449609 82.81725806347009 0.0) (-0.9026499999999942 -1.052371999999991 0.0))
number:	nil
threat:	nil
snode:	<snode 1072948>
pattern:	nil
duration:	one-shot
position:	nil
goalptr:	nil
conditions:	nil
ksarptr:	nil

Each of these subgoals points to the parent track as the source, and to the supporting segment node as the snode. The KSAR generated from each of these subgoals will activate the VERIFY KS. This KS is part of the BB process, i.e., not spun-off as a separate process. If the segment is reverified to be in the same track grouping, then nothing is done, except recording the verification result by removing the node from the *checklyst*. If the segment is not reverified, then the KS does three things. First, the KS removes the segment pointers in the track node, i.e., the pointers to the siblings or branches of the tree. Second, the KS removes the parent pointer in the segment node or the pointer to the root of the tree

representing the track. And third, the KS removes the pointer from the *checklyst* of the track node.

The snode orphaned by the VERIFY KS is the following segment node.

```
<snode 1073184> is an instance of flavor snode with instance variables:
   type:        segment
   time:        (13 12 11 10 9 8 7 6 5 4 3 2 1 0)
   coord:       ((89.09235 84.915828 0.0) (90.0064 85.935872 0.0)
                (90.90904999999999 86.98824399999999 0.0)
                (91.8 88.074 0.0) (92.67895 89.19419600000001 0.0)
                (93.54559999999999 90.34988800000001 0.0)
                (94.39964999999999 91.542132 0.0)
                (95.24079999999999 92.771984 0.0)
                (96.06874999999999 94.04049999999999 0.0)
                (96.8832 95.348736 0.0) (97.68385000000001 96.697748 0.0)
                (98.4704 98.08859200000001 0.0)
                (99.24254999999999 99.522324 0.0) (100.0 101.0 0.0))
   number:      14
   cpa:         (7.210431076363022 -6.461186503081834 0.0)
   linear:      ((89.09235 84.915828 0.0)
                (-0.9140500000000031 -1.020043999999999 0.0))
   tnode:       nil
   threat:      nil
```

The BB detects the unmatched segment node, then constructs a distinct track for this segment; the resulting solution is the two track nodes given below. The first track node is the newly created node from the unmatched segment node. The second track node is the old, established track node that now contains only two supporting segment nodes. The solution of the tracking problem is now two trees (and, in general, a forest of trees) that represent two separate tracks. Track 1 of Figure A-6 is as follows:

```
<tnode 1074280> is an instance of flavor tnode with instance variables:
   type:           track
   time:           (13)
   last-coord:     (89.09235 76.75794999999999 0.0)
   last-velocity:  (-0.9140500000000031 -1.382850000000005 0.0)
   threat:         nil
   snode:          (<snode 1072948>)
   cpa-bracket:    ((20.45362766455384 32.93339536190769)
                    (-27.08435313612137 -8.203934384099306))
   check:          nil
   checklyst:      nil
```

Track 2 of Figure A-6 is as follows:

```
<tnode 1074676> is an instance of flavor tnode with instance variables:
   type:           track
   time:           (13)
   last-coord:     (89.09235 84.4169265 0.0)
   last-velocity:  (-0.9140500000000031 -1.019809500000001 0.0)
   threat:         t
   snode:          (<snode 1073144> <snode 1073184>)
   cpa-bracket:    ((-0.977760816000675 15.82484430129828)
                    (-15.96939768539792 2.67651494722635))
   check:          nil
   checklyst:      nil
```

EXAMPLE 4

This example shows three craft, one of which has a signal that fades for a short period. This example illustrates how faded segments may be matched up with established segments, and illustrates a different type of goal to activate the extend-segments KS.

In Section 6 of this report, we showed that the MERGE-SEGMENTS KS is activated by a recurrent goal. This means that once a KSAR is created and scheduled, the goal is inhibited from creating another KSAR until the KS finishes its attempt to extend atrophied segments. In this example, three trajectories, as illustrated in Figure A-7, are shown. Only one of these trajectories fades. The segment node of this trajectory is given by

```
<snode 1073144> is an instance of flavor snode with instance variables:
   type:        segment
   time:        (2 1 0)
   coord:       ((1.47 98.4704 0.0)
                (0.7425 99.24254999999999 0.0)
                (0.0 100.0 0.0))
   number:      3
   cpa:         (49.92671489395662 47.0396750441671 0.0)
   linear:      ((1.47 98.4704 0.0)
                (0.7274999999999999 -0.7721499999999963 0.0))
   tnode:       <tnode 1074676>
   threat:      nil
```

The initial part of the trajectory forms a track node. However, after the time stamp of 2 the trajectory input fades and a time gap for the input values results. The path does not return until the time stamp of 8. At this time, a new segment node is generated on the BB. This new segment node represents a newly found segment that looks as follows:

```
<snode 1075068> is an instance of flavor snode with instance variables:
   type:          segment
   time:          (8 7)
   coord:         ((5.52 93.54559999999999 0.0)
                  (4.8825 94.39964999999999 0.0))
   number:        2
   cpa:           (48.38657378899811 36.11783945961739 0.0)
   linear:        ((5.52 93.54559999999999 0.0)
                  (0.6374999999999993 -0.8540500000000009 0.0))
   tnode:         <tnode 1074172>
   threat:        nil
```

Note that both these segments point to a track node, so that a track exists for each one. Moreover, these tracks are different because the second segment was not determined to be an extension of the first one at this time.

After the MERGE KS has run, the second segment is recognized as an extension of the first segment. So, the latest track and segment nodes are retained, the older segment is removed from the BB, and the pointer from the track of that segment node is deleted. If the removed segment is the only segment supporting that track, then the entire track is removed by removing that node from the BB, as is the case here.

The KSAR that initiates this KS looks as follows:

```
<bbevent1071572> is an instance of flavor bbevent with instance
variables:
   source:        nil
   action:        nil
   type:          extend-segments
   variable:      nil
   time:          nil
   coord:         nil
   number:        nil
   threat:        nil
   snode:         nil
   pattern:       nil
   duration:      recurrent
   position:      nil
   goalptr:       nil
   conditions:    nil
   ksarptr:       <ksar 1073224>
```

The KS has a recurrent duration and contains a pointer to the KSAR that initiates MERGE-SEGMENTS.

The resulting track node, which was established for the reappearing track, now represents both the current and the merged tracks. The older segment and its track have been removed and are now represented by this track node as well. The track node looks as follows:

```
<tnode1074632> is an instance of flavor tnode with instance variables:
   type:           track
   time:           (8)
   last-coord:     (5.52  93.54559999999999  0.0)
   last-velocity:  (0.637499999999993  -0.8540500000000009  0.0)
   threat:         nil
   snode:          (<snode 1075068>)
   cpa-bracket:    ((44.0999164100983  52.67323116789792)
                    (30.37506340557913  41.86061551365565))
   check:          nil
   checklyst:      nil
```

The old segment has been patched to the new track, although the old segment data have not been appended to the new track. No history of this older track is included in the current track because the segment nodes and hit nodes are removed from the BB as soon as possible. However, a short history trail could easily be added to the track node.

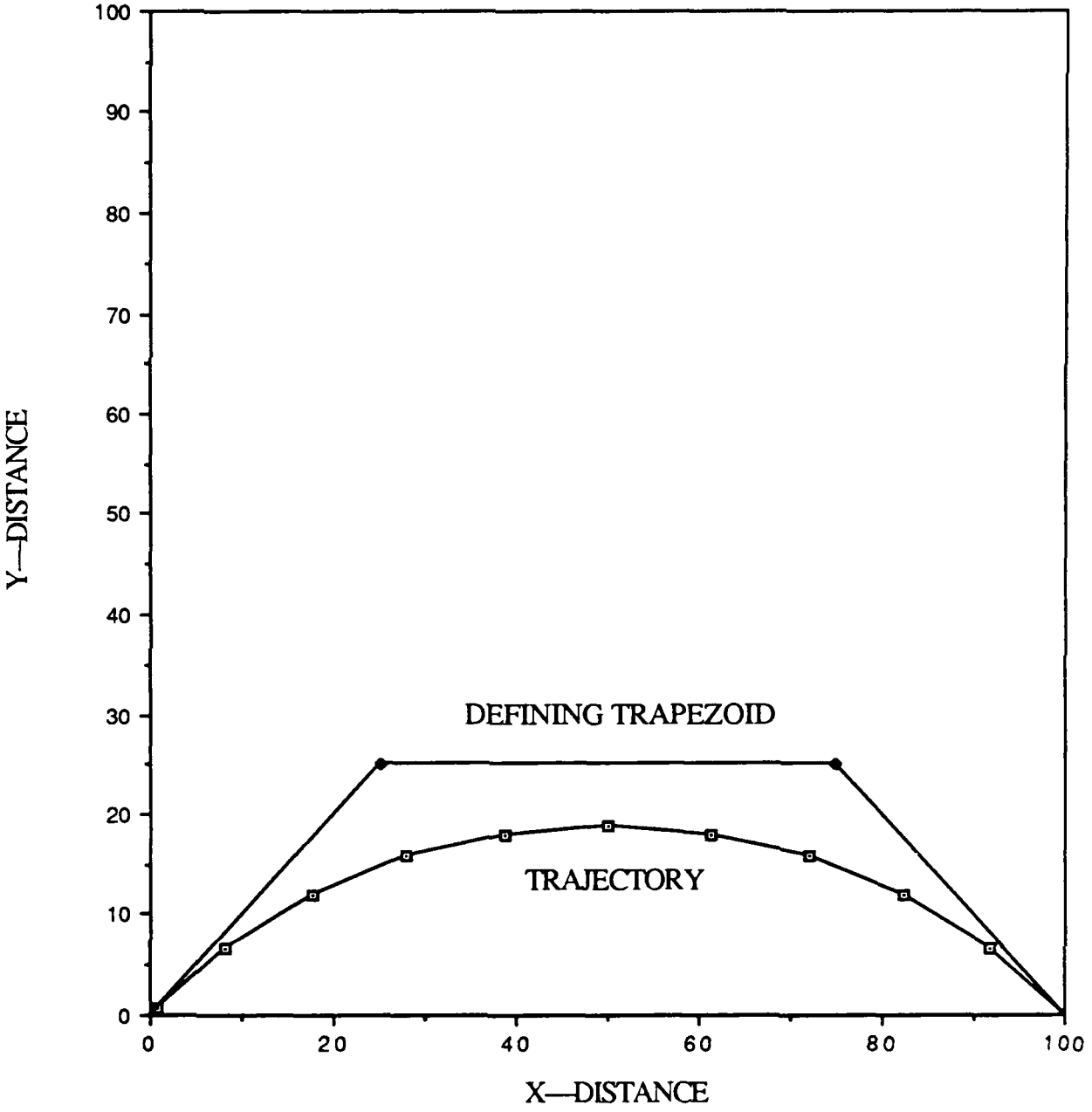


FIGURE A—1. Example 1 Shows Single Trajectory Generated Via Bezier's Curve With Every Tenth Point Shown. *Defining trapezoid* shown with curve.

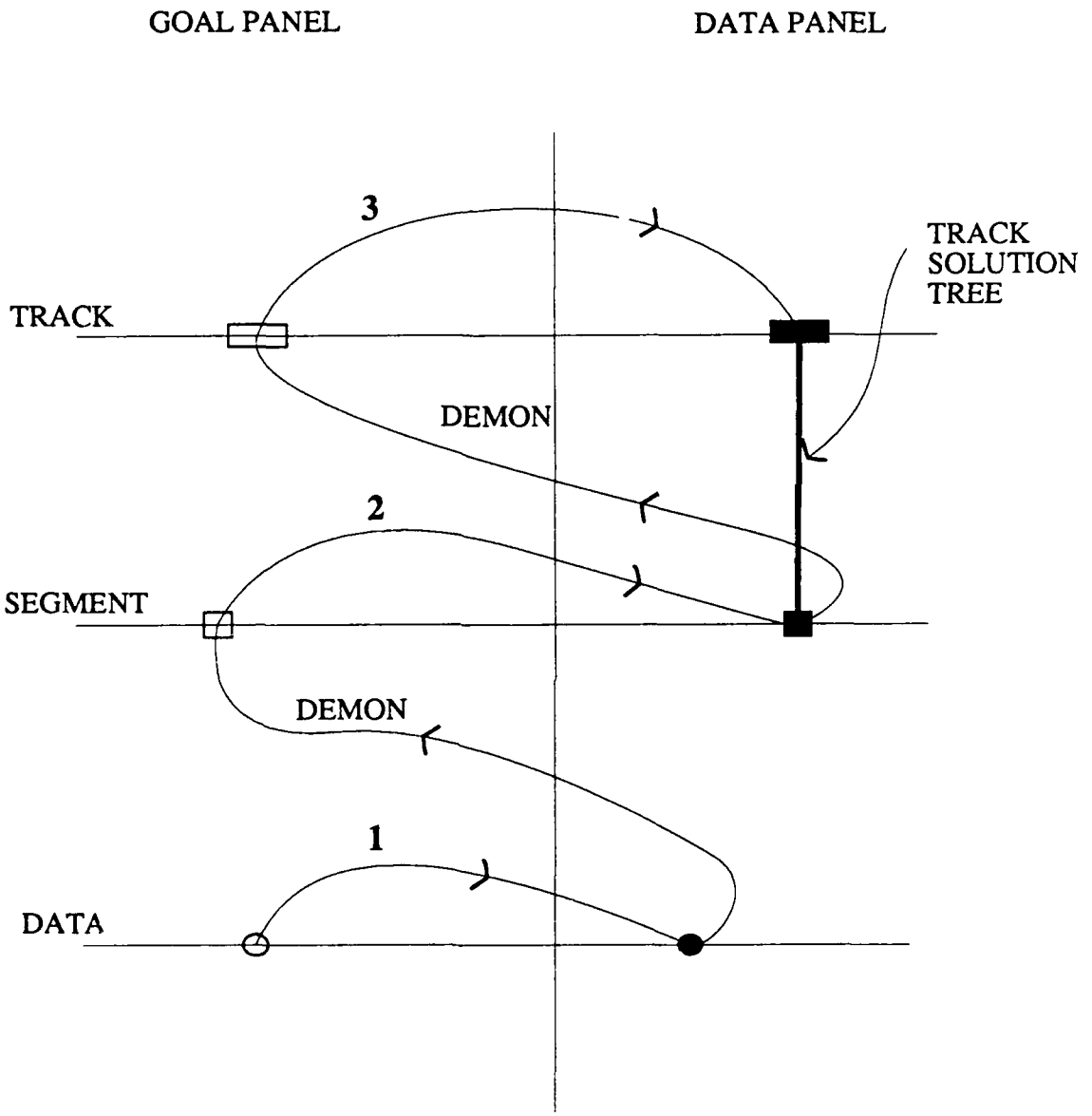


FIGURE A—2. Example 1 BB Trace Shows Track Formation for a Single Trajectory.

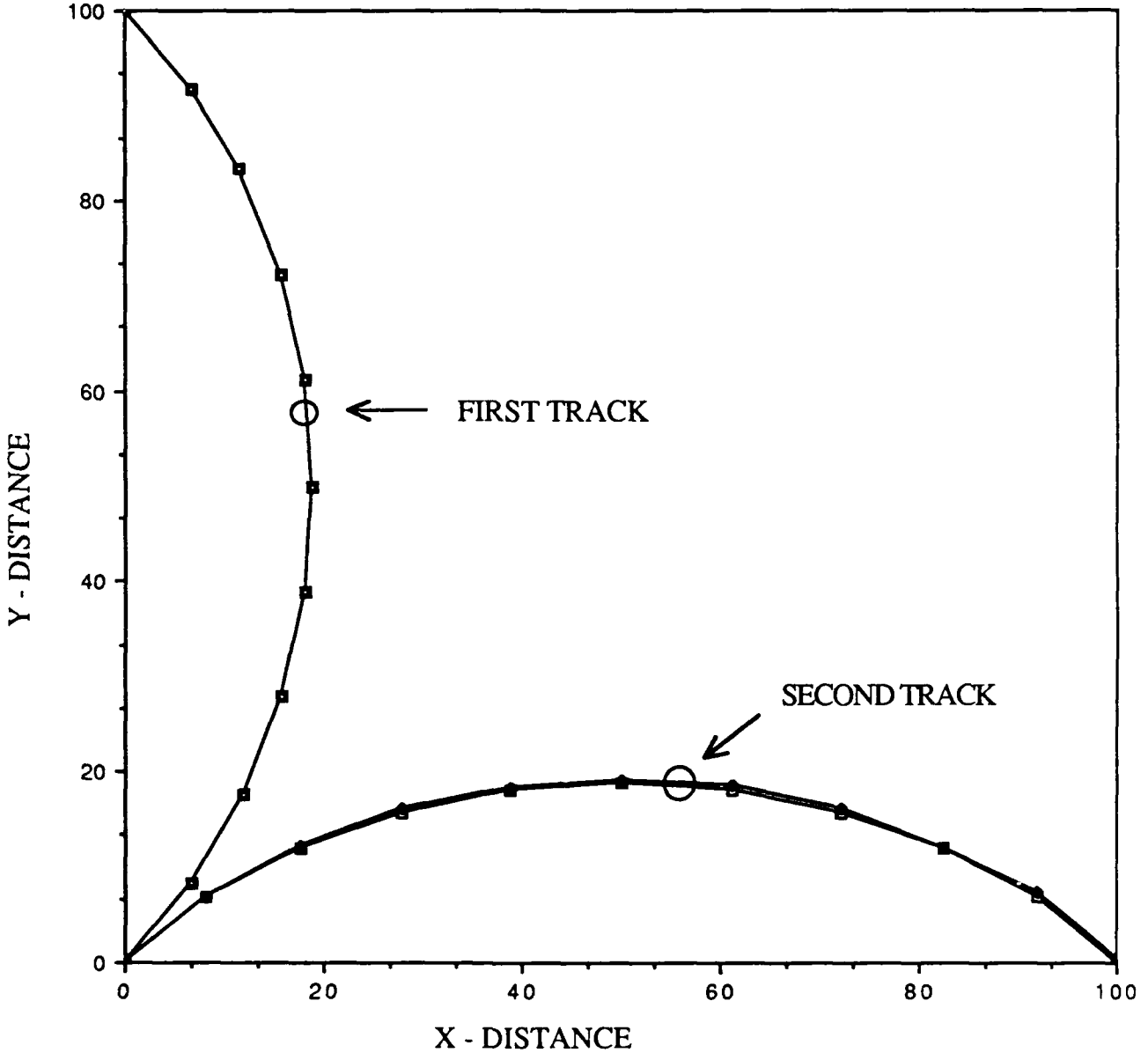


FIGURE A—3. Example 2 Trajectory Shows Three Separate Craft With Two Tracks Indicated.

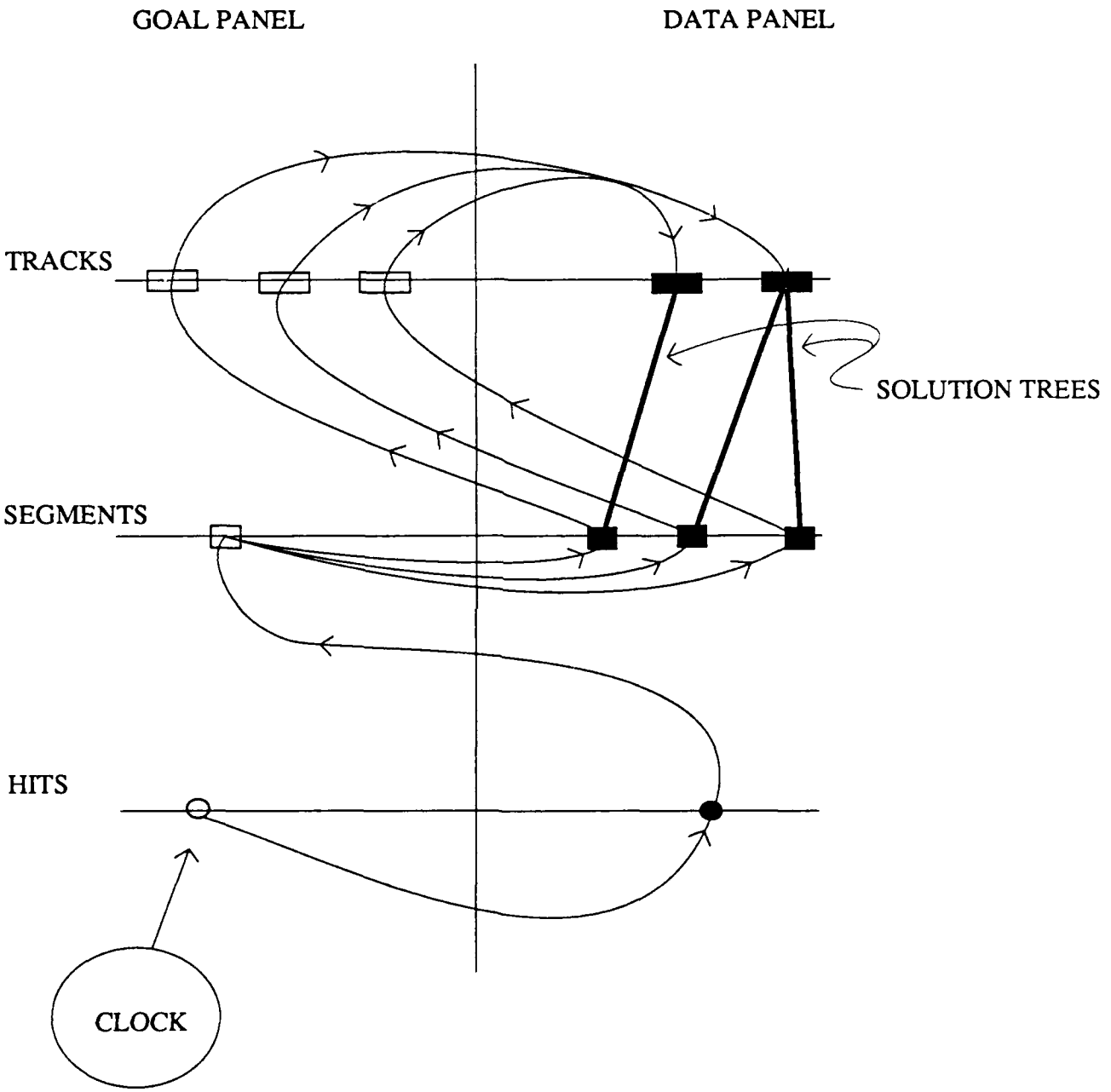


FIGURE A—4. Example 2 BB Trace Shows BB for the Three-Trajectory, Two-Track Example With Two Solution Trees.

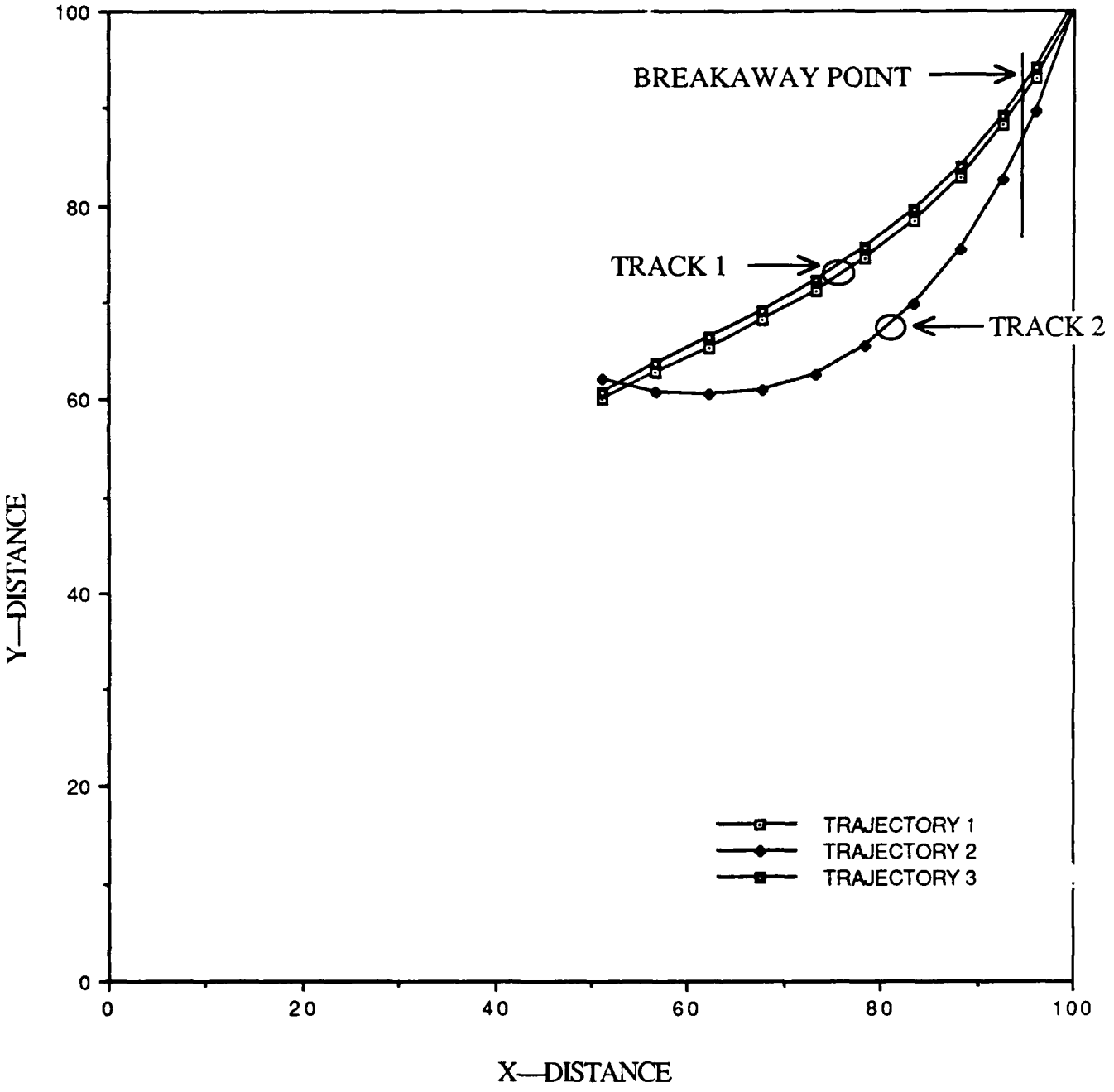


FIGURE A—5. Example 3 Trajectory Shows Three-Aircraft Problem Where One Track Breaks Away.

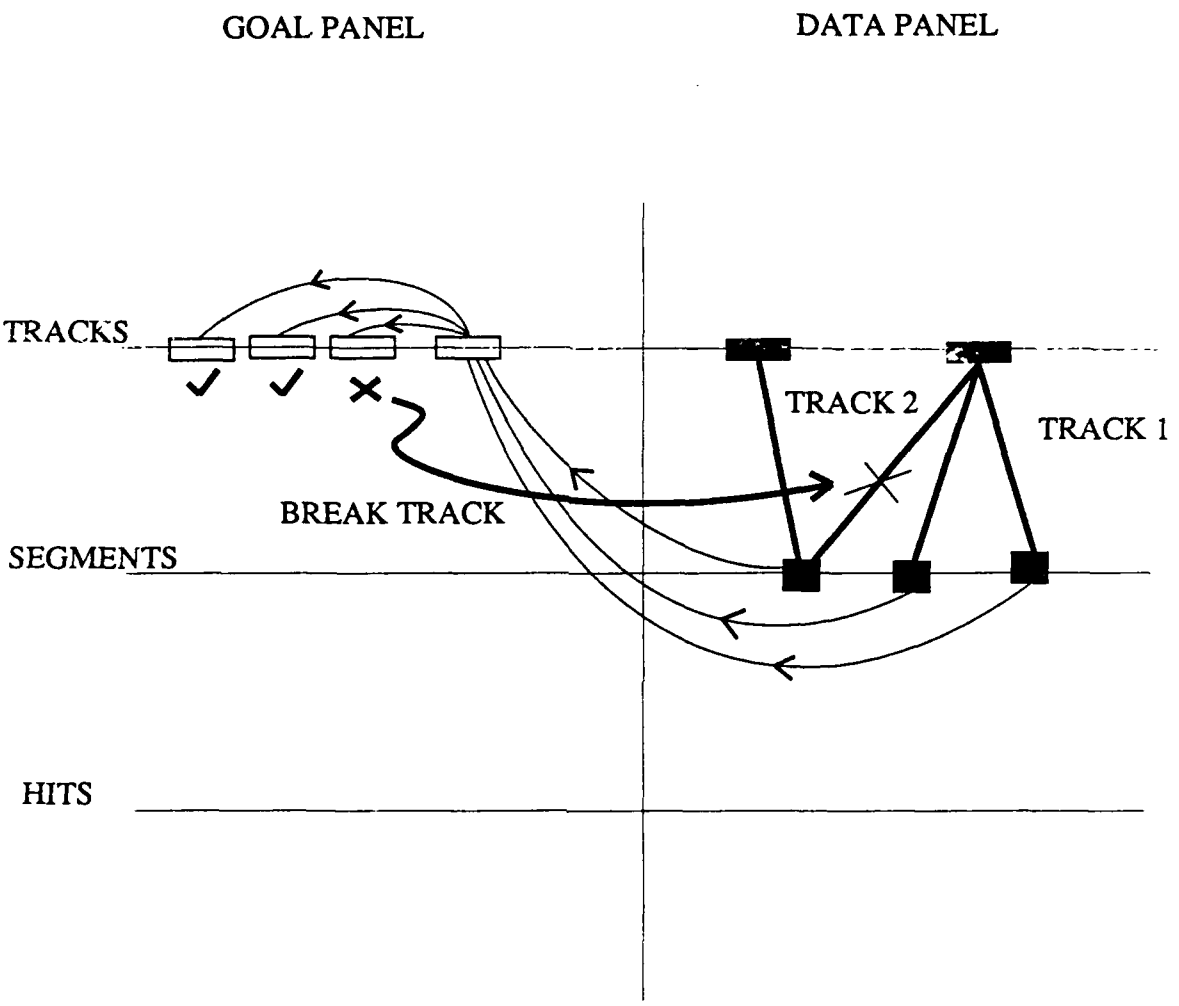


FIGURE A—6. Example 3 BB Trace Illustrates How Support of the Track—1 Hypothesis by the Segment Node is Broken. Segment node eventually will support new track hypothesis. Subgoaling triggered by failure of spline test is illustrated in goal panel.

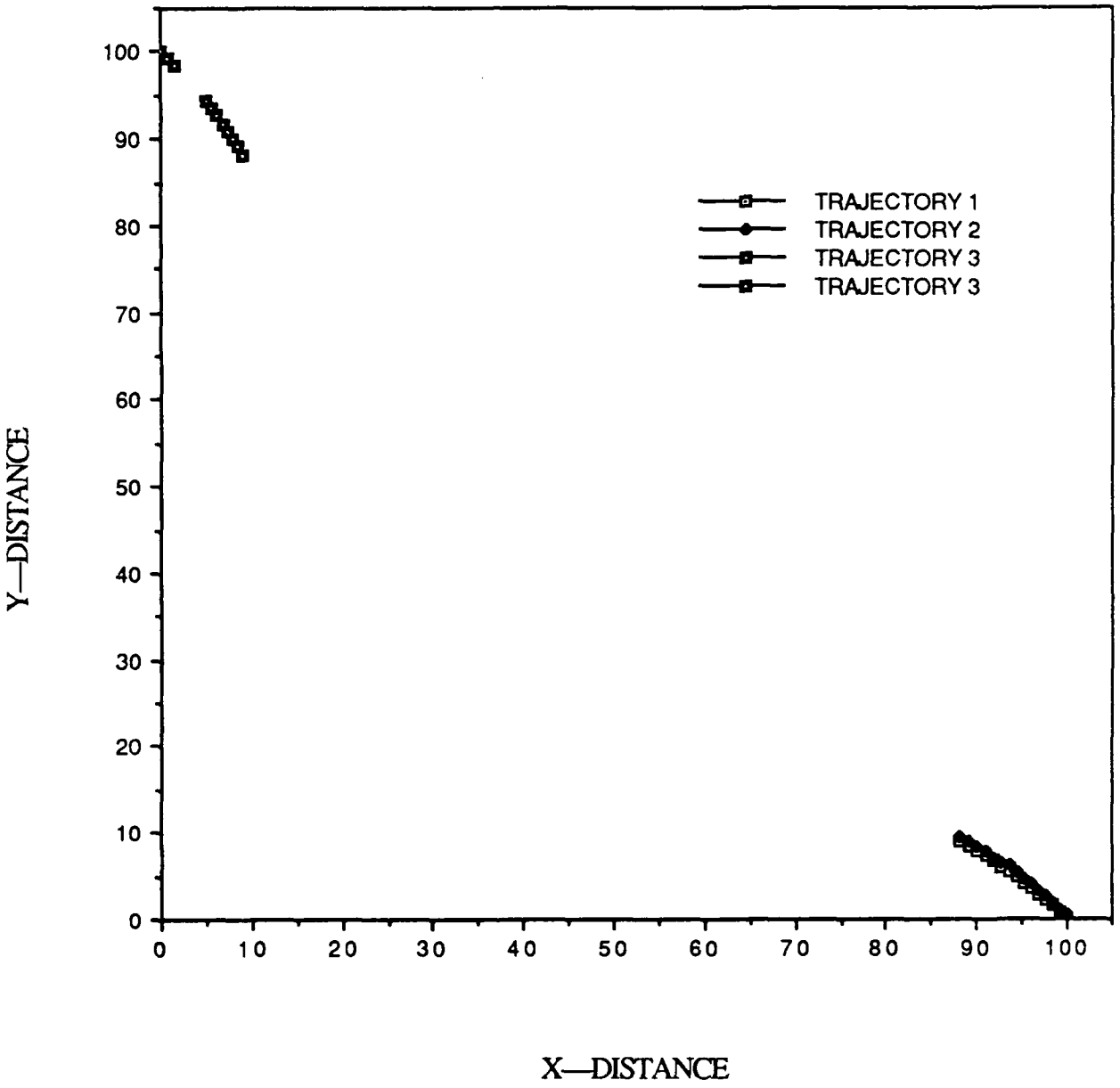


FIGURE A—7. Example 4 Trajectory Illustrates the Merging of Two Segments.